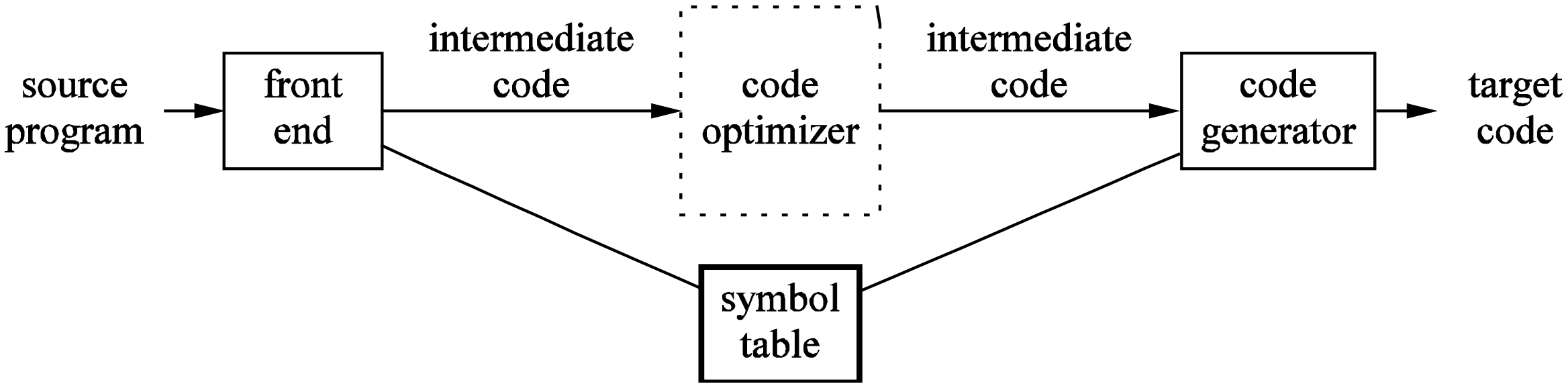
Code Generation

- The final phase is the code generator.
- The requirements on a code generator are severe.



Motivation

Requirements

- Preserve semantic meaning of source program
- Make effective use of available resources of target machine
- Code generator itself must run efficiently

Challenges

- Problem of generating optimal target program is undecidable
- Many sub-problems encountered in code generation are computationally intractable

Main Tasks of Code Generator

Input: IR

Output: Target Program

- Instruction selection
  choosing appropriate target-machine instructions to implement the IR statements
- Registers allocation and assignment
  deciding what values to keep in which registers
- Instruction ordering
  deciding in what order to schedule the execution of instructions

Design Issues of a Code Generator: Input

- The input to the code generator consists of:
  - Intermediate code produced by the front end (and perhaps optimizer)
    - Remember that intermediate code can come in many forms
      - three-address presentations (quadruples, triples, …)
      - Virtual machine presentations (bytecode, stack-machine, …)
      - Linear presentation (postfix, …)
      - Graphical presentation (syntax trees, AST, DAGs,…)
    - We are concentrating on three-address code and AST
    - but techniques apply to other possibilities as well
  - Information in the symbol table (used to determine run-time addresses of data objects)
- The code generator typically assumes that:
  - The input is free of errors
  - Type checking has taken place and necessary type-conversion operators have already been inserted

Design Issues of a Code Generator: Target Program

- The target program is the output of the code generator
- Can take a variety of forms
  - instruction set architecture (RISC, CISC)
  - absolute machine language
  - relocatable machine language
    - Can compile subprograms separately
    - Added expense of linking and loading
  - assembly language
    - Makes task of code generation simpler
    - Added cost of assembly phase

Design Issues of a Code Generator: Instruction Selection

- The complexity of mapping IR program into code-sequence for target machine depends on:
  - Level of IR (high-level or low-level)
  - Nature of instruction set (data type support)
  - Desired quality of generated code (speed and size)
- If efficiency is not a concern, instruction selection is straightforward
  - For each type of three-address statement, there is a code skeleton outlines target code
  - Example, `x := y + z`, where `x`, `y`, and `z` are statically located, can be translated as:

```
MOV R0,y /* load y into register R0 */
ADD R0,R0,z /* add z to R0 */
MOV x,R0 /* store R0 into x */
```

Design Issues of a Code Generator: Instruction Selection

- Often, the straightforward technique produces poor code:

```
a := b + c
d := a + e
```

⟹

```
MOV R0, b
ADD R0,R0,c
MOV a, R0
MOV R0, a
ADD R0,R0,e
MOV d, R0
```

- A naïve translation may lead to correct but unacceptably inefficient target code:

```
a := a + 1
```

⟹

```
MOV R0, a
ADD R0,R0,#1
MOV a, R0
```

(efficient code: INC a)

Design Issues of a Code Generator:

Register Allocation

- Selecting the set of variables that will reside in registers at each point in the program

Register Assignment

- Picking the specific register that a variable will reside in

Design Issues of a Code Generator: Evaluation Order

- Selecting the order in which computations are performed
- Affects the efficiency of the target code
- Some orders require fewer registers than others
- Picking a best order is NP-complete

Compilation Schemes

- AST to MIPS programs (For PA4, used by JVM)
  - AST to Stack machines
  - Stack machines to MIPS assembly language (RISC)
  - Object Oriented Code Generation
    - ✓ Object Layout, Dynamic Dispatch
- Three-address to a Simple Target-Machine (CISC-like)
- Both of them can be generalized to other target-machines and IRs

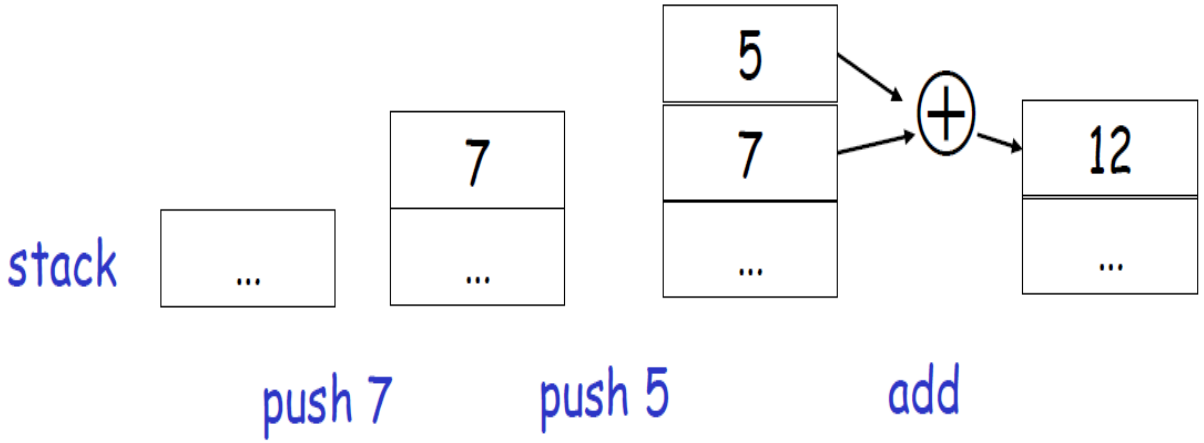
Stack Machines

- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results

push i - place the integer i on top of the stack
add - pop two elements, add them and put the result back on the stack

A program to compute 7 + 5:

```
push 7
push 5
add
```



Why Use a Stack Machine ?

- Each operation takes operands from the same place and puts results in the same place
  - Location of the operands is implicit, always on the top of the stack
  - No need to specify operands explicitly
  - No need to specify the location of the result
  - Smaller encoding of instructions
  - More compact programs

  e.g. Instruction "add" as opposed to "add r_1, r_2"
- This means a uniform compilation scheme, therefore a simpler compiler
  - This is what you have to do for PA4
- This is one reason why Java Bytecodes use a stack evaluation model

Optimizing the Stack Machine

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
- Idea: keep the top of the stack in a register (called accumulator)
  - Register accesses are faster

- The "add" instruction is now

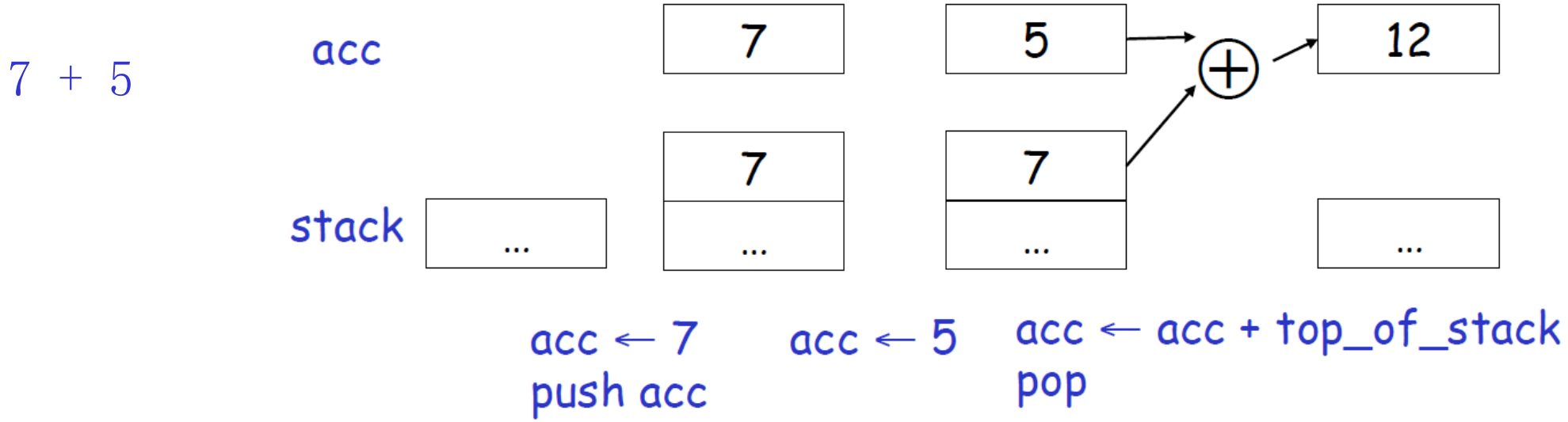
$$acc \leftarrow acc + top_of_stack$$

  - Only one memory operation!

Stack Machine with Accumulator

Invariants

- The result of computing an expression is always in the accumulator
- For an operation $op(e_1,\ldots,e_n)$ push the accumulator on the stack after computing each of $e_1,\ldots,e_{n-1}$
  1. The result of e_n is in the accumulator before op
  2. After the operation pop n-1 values
- After computing an expression the stack is as before



Example

Given the stack machine code for: 3 + (7 + 5)

Code	Acc	Stack
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>

It is **very important** that the stack is preserved across the evaluation of a subexpression

- Stack before the evaluation of 7 + 5 is 3, <init>
- Stack after the evaluation of 7 + 5 is 3, <init>
- The first operand is on top of the stack

From Stack Machines to MIPS

- The compiler generates code for a stack machine with accumulator
- We want to run the resulting code on the MIPS processor (or simulator)
- We implement stack machine instructions using MIPS instructions and registers
- The accumulator is kept in MIPS register $a0
- The stack is kept in memory
- The stack grows towards lower addresses
  - Standard convention on the MIPS architecture
- The address of the next location on the stack is kept in MIPS register $sp
  - The top of the stack is at address $sp + 4

MIPS architecture

- Prototypical Reduced Instruction Set Computer (RISC) architecture
- Arithmetic operations use registers for operands and results
- Must use load and store instructions to use operands and results in memory
- 32 general purpose registers (32 bits each)
  - We will use $sp for stack pointer
  - $a0 for accumulator
  - $t1 for a temporary register,
  - Using $a0 and $t1 is just a convention. You could pick $a2,$a3,$t0, etc..
- Read the SPIM_Manual for more details

A Sample of MIPS Instructions

- lw reg_1 offset(reg_2)
  - Load 32-bit word from address reg_2 + offset into reg_1
- add reg_1 reg_2 reg_3
  - $reg_1 \leftarrow reg_2 + reg_3$
- sw reg_1 offset(reg_2)
  - Store 32-bit word in reg_1 at address reg_2 + offset
- addiu reg_1 reg_2 imm
  - $reg_1 \leftarrow reg_2 + imm$
  - "u" means overflow is not checked
- li reg imm
  - reg ← imm

Example

- The stack-machine code for 7 + 5 in MIPS:

acc ← 7	li $a0 7
push acc	sw $a0 0($sp)
	addiu $sp $sp -4
acc ← 5	li $a0 5
acc ← acc + top_of_stack	lw $t1 4($sp)
	add $a0 $a0 $t1
pop	addiu $sp $sp 4

We now generalize this to a simple language…

Abbreviations

- We define the following abbreviations

push $t	sw $t 0($sp) addiu $sp $sp -4
pop	addiu $sp $sp 4
$t ← top	lw $t 4($sp)

A Small Language

- A language with integers and integer operations

$$
\begin{array}{lll}
P & \rightarrow & D;\ P \mid D \\
D & \rightarrow & \text{def id(ARGS)} = E; \\
ARGS & \rightarrow & \text{id, ARGS} \mid \text{id} \\
E & \rightarrow & \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4 \\
 & & \mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)
\end{array}
$$

- The first function definition f is the "main" routine
- Running the program on input i means computing f(i)
- Program for computing the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
                if x = 2 then 1 else fib(x - 1) + fib(x – 2)
```

Code Generation Strategy

- For each expression e we generate MIPS code that:
  1. Computes the value of e in $a0
  2. Preserves $sp and the contents of the stack
- We define a code generation function cgen(e) whose result is the code generated for e

Code Generation for Constants

- The code to evaluate a constant simply copies it into the accumulator:

cgen(i) = li $a0 i

- Note that this also preserves the stack, as required

Code Generation for Add/Sub

cgen(e_1 +/- e_2) =

```
cgen(e1)
push $a0
cgen(e2)
$t1 ← top
add/sub $a0 $t1 $a0
pop
```

Is it possible optimization: Put the result of e_1 directly in register $t1 ?

Code Generation for Add. Wrong!

- Optimization: Put the result of e_1 directly in $t1?

```
cgen(e1 + e2) =
        cgen(e1)
        mov $t1 $a0
        cgen(e2)
        add $a0 $t1 $a0
```

- Try to generate code for : 3 + (7 + 5)

Code Generation for Conditional

- We need flow control instructions if $e_1 = e_2$ then e_3 else e_4
- New instruction: beq reg1 reg2 label
  - Branch to label if reg1 = reg2
- New instruction: b label
  - Unconditional jump to label

```
cgen(if e1 = e2 then e3 else e4) =
        cgen(e1)
        push $a0
        cgen(e2)
        $t1←top
        pop
        beq $a0 $t1 true_branch
false_branch:
    cgen(e4)
    b end_if
true_branch:
    cgen(e3)
end_if:
```

The Activation Record

- Code for function calls and function definitions depends on the layout of the activation record

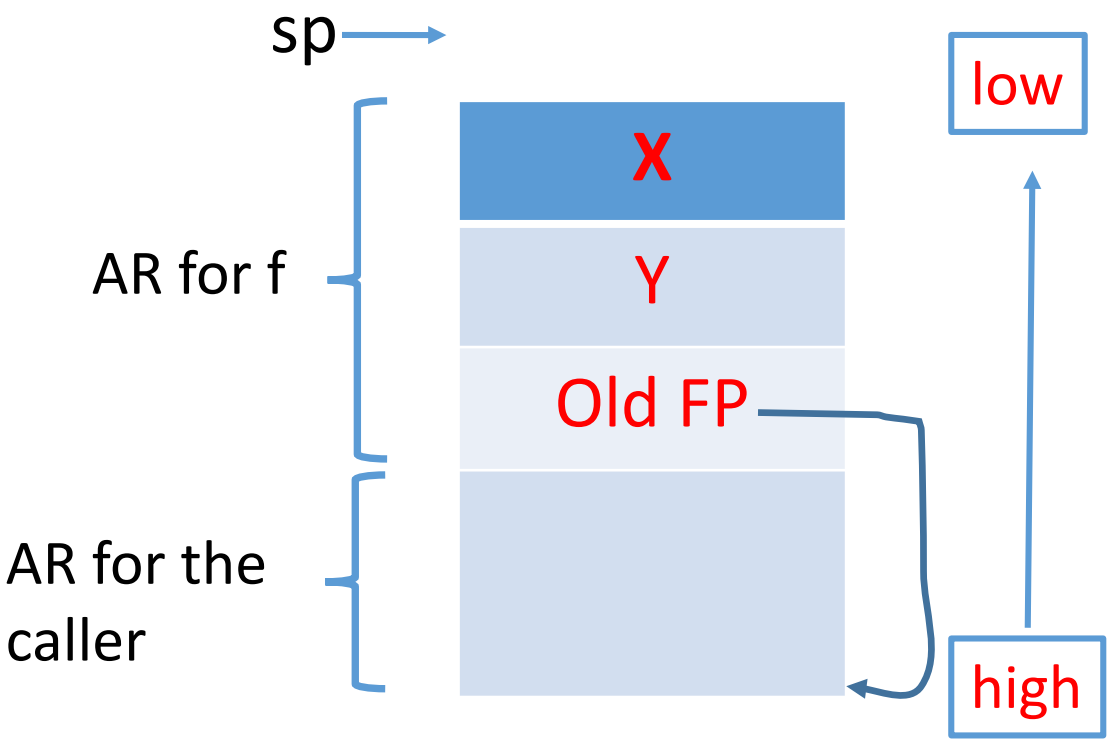
- A very simple AR suffices for this language:
  - The result is always in the accumulator
  - No need to store the result in the AR
  - The activation record holds actual parameters
  - For $f(x_1,\ldots,x_n)$ push $x_n,\ldots,x_1$ on the stack

The Activation Record (Cont.)

- The stack discipline guarantees that on function exit $sp is the same as it was on function entry
  - No need to save $sp
- We need the return address
- It's handy to have a pointer to start of the current activation
  - This pointer lives in register $fp (frame pointer)
  - Reason for frame pointer will be clear shortly
- Summary: For this language, an AR with
  1. the caller's frame pointer $fp
  2. the actual parameters
  3. and the return address

Example

- Consider a call to f(x,y),
- The AR will be:



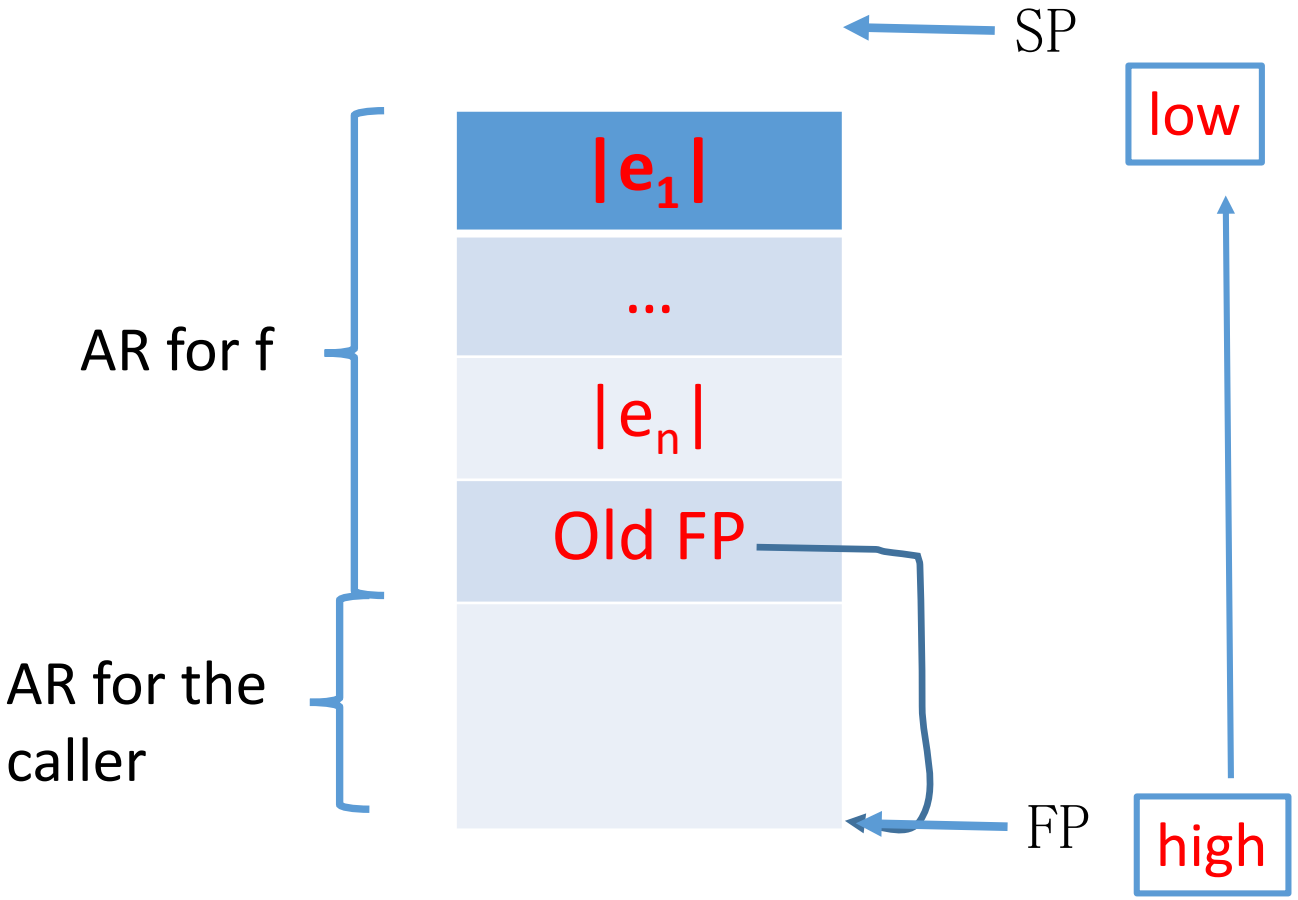
Code Generation for Function Call

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation
- New instruction: jal label // call label
  - –Jump to label and save address of next instruction in $ra (one step),
  - – On other architectures the return address is stored on the stack by the "call" instruction, i.e., push return address onto the stack and then jump to the entry point of the function (two steps)

Jump to the entry point of the function, then push return address onto the stack (two steps) Why not?

Code Generation for Function Call (Cont.)

```
cgen( f(e1,…,en) ) =
        push $fp
        cgen(en)
        push $a0
        …
        cgen(e1)
        push $a0
        jal f_entry
```

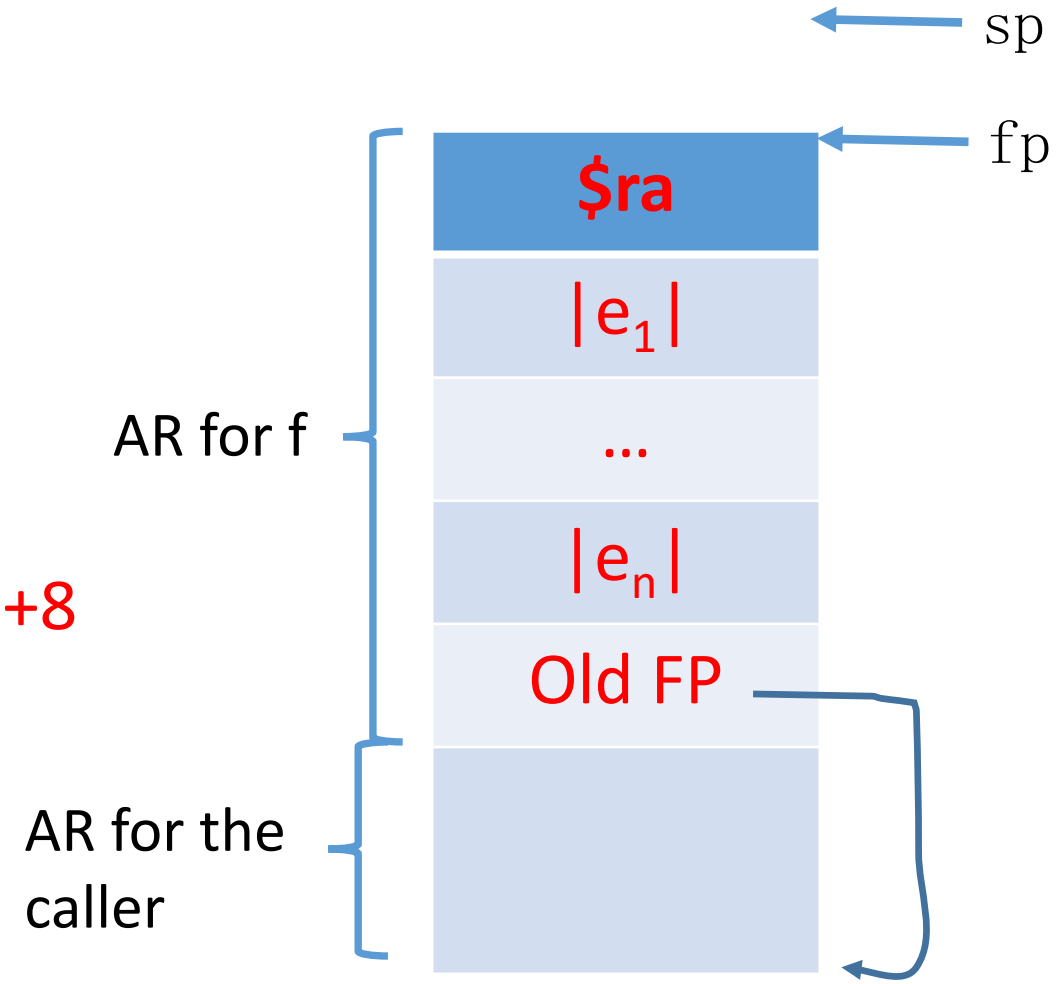


- The caller saves the return address in register $ra
- The AR so far is 4*n+4 bytes long

Code Generation for Function Definition

cgen(def f(x_1,…,x_n) = e) =

```
f_entry:
mov $fp $sp
push $ra
cgen(e)
$ra ← top
addiu $sp $sp 4*n+8
lw $fp 0($sp)
b $ra
```



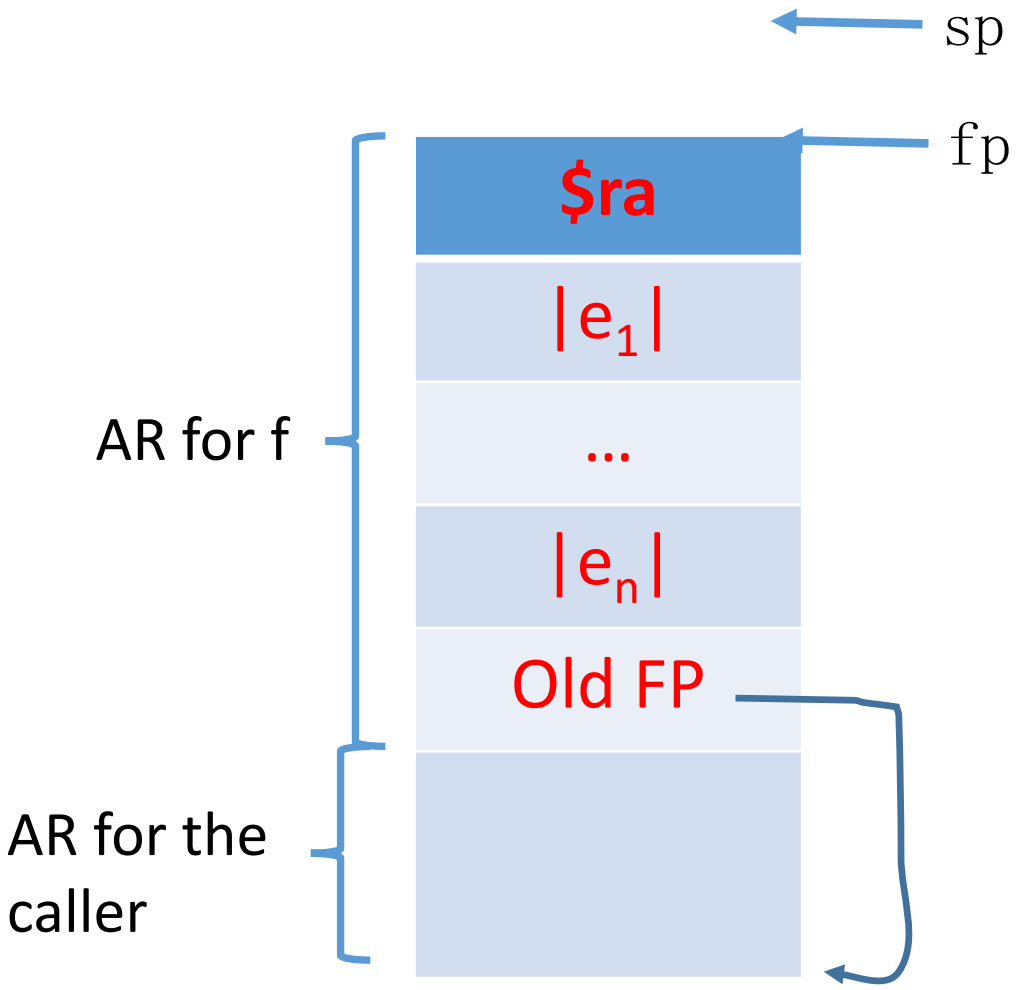
cgen(f(e_1,…,e_n)) =

```
push $fp
cgen(e_n)
push $a0
…
cgen(e_1)
push $a0
jal f_entry
```

4*n+4

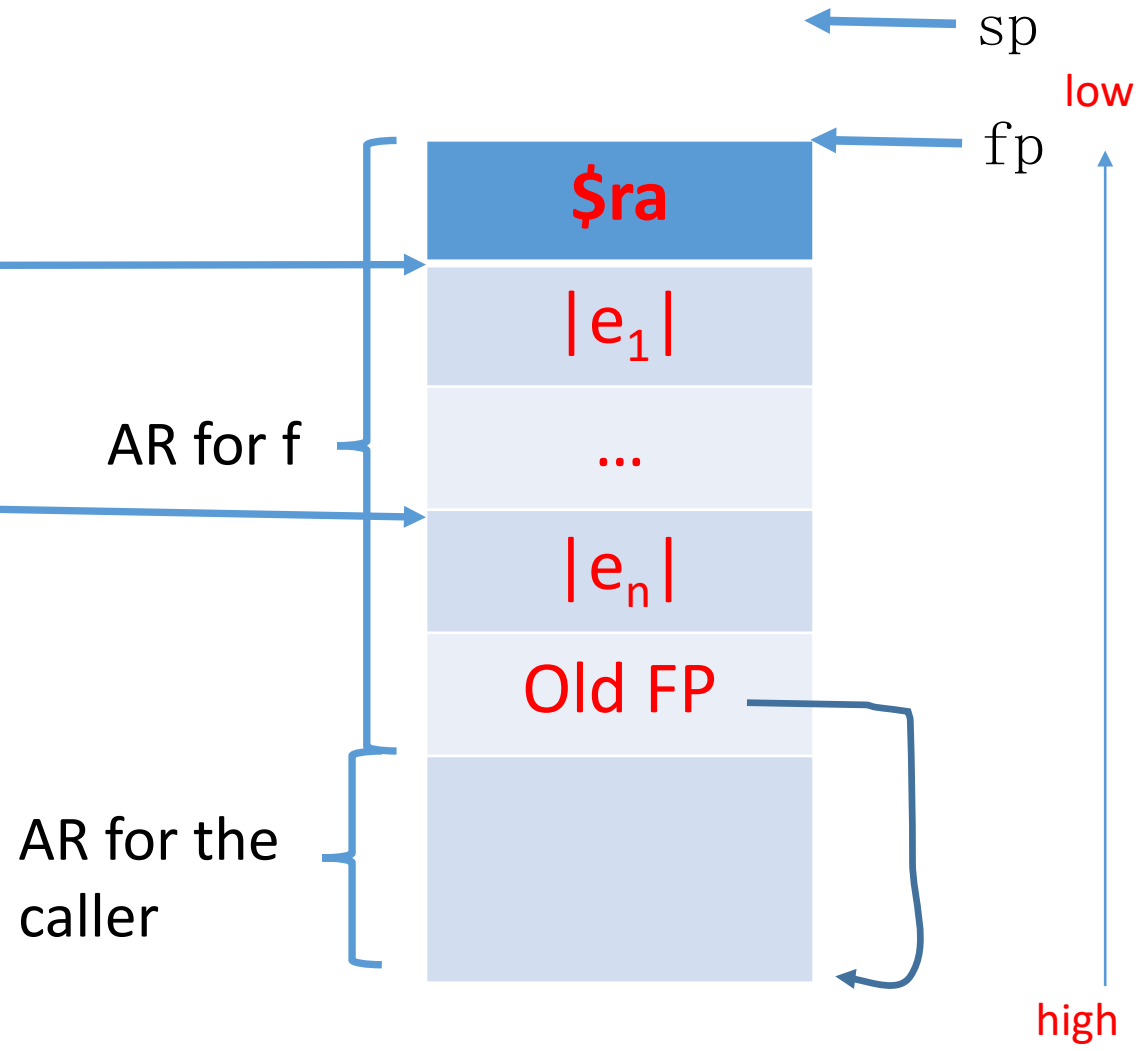
Code Generation for Variables

- Variable references are the last construct
- The "variables" of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller
- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp
- Solution: use a frame pointer $fp
  - Always points to the return address on the stack
  - Since it does not move it can be used to find the variables



Code Generation for Variables

- Function: $f(x_1,\ldots,x_n)=e$
- formal parameters
  - ✓ x_1
  - ✓ …
  - ✓ x_n



cgen(x_i) = lw $a0 z($fp)

where (z = 4*i)

cgen(x_2) = lw $a0 8($fp)

Summary

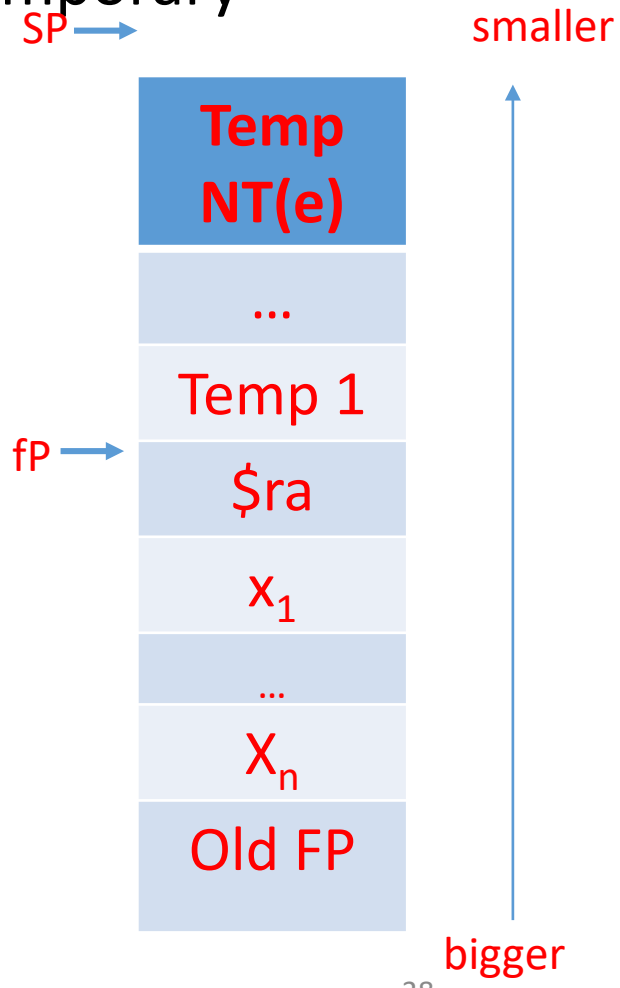
- Stack-machine scheme for code generation
  - Very simple but very slow code (Why)
  - Storing/loading temporaries requires a store/load and $sp adjustment
- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- We recommend you use a stack machine for your Cool compiler (it's simple)
- Production compilers do different things
  - Emphasis is on keeping values (esp. current stack frame) in registers
  - Intermediate results are laid out in the AR, not pushed and popped from the stack

Allocating Temporaries in the AR

- Idea: Keep temporaries in the AR avoiding load and store
- The code generator must assign a location in the AR for each temporary

$NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$
$NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$
$NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$
$NT(f(e_1,\ldots,e_n)) = \max(NT(e_1),\ldots,NT(e_n))$
$NT(\text{int}) = 0$
$NT(\text{id}) = 0$

- For a function definition $f(x_1,\ldots,x_n) = e$ the AR has $2 + n + NT(e)$ elements
  - Return address
  - Frame pointer
  - n arguments
  - NT(e) locations for intermediate results

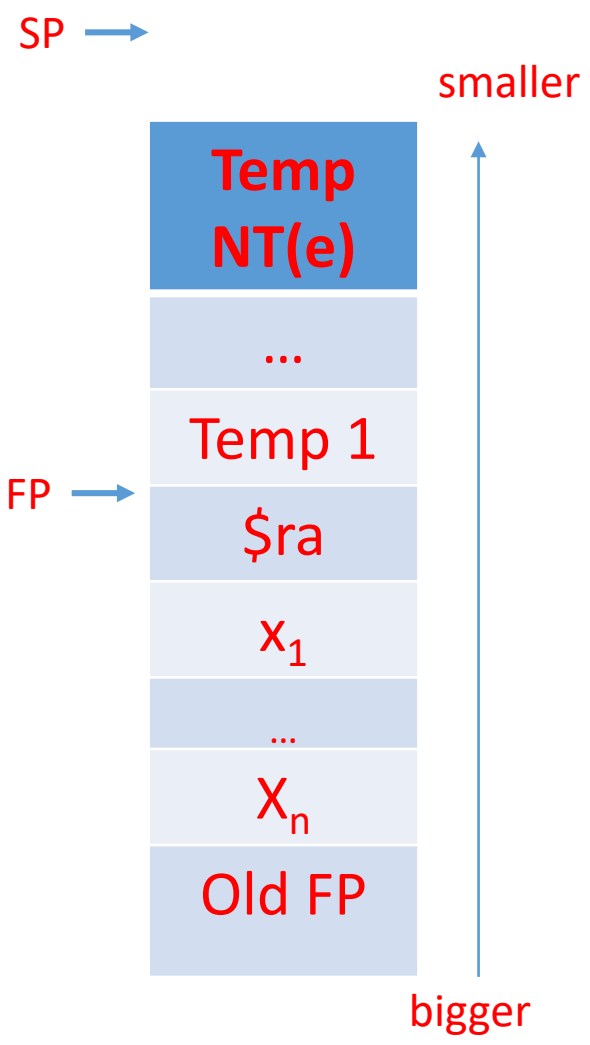


Revised Code Generation

- Code generation must know how many temporaries are in use at each point

- Add a new argument to code generation: the position of the next available temporary

cgen(e, n) : generate code for e and use temporaries whose address is ($fp - n) or lower



Code Generation for +

```
cgen(e1 + e2, n) =
        cgen(e1, n)
        sw $a0  -n($fp)
        cgen(e2, n + 4)
        lw $t1 -n($fp)
        add $a0 $t1 $a0
```

```
cgen(e1 + e2) =
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
        cgen(e2)
        lw $t1 4($sp)
        add $a0 $t1 $a0
        addiu $sp $sp 4
```

Code Generation for Object-Oriented Languages

- OO implementation = Stuff from last lecture + More stuff
- OO Slogan: If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected
- This means that code in class A works unmodified for an object of class B

- **Two Issues**
  - How are objects represented in memory?
  - How is dynamic dispatch implemented?

Object Layout

An object is like a struct in C. The reference

foo.field

is an index into a foo struct at an offset corresponding to field

Objects in Cool are implemented similarly

- Objects are laid out in contiguous memory
- Each attribute stored at a fixed offset in object
- When a method is invoked, the object is self and the fields are the object's attributes

Cool Object Layout (Cont.)

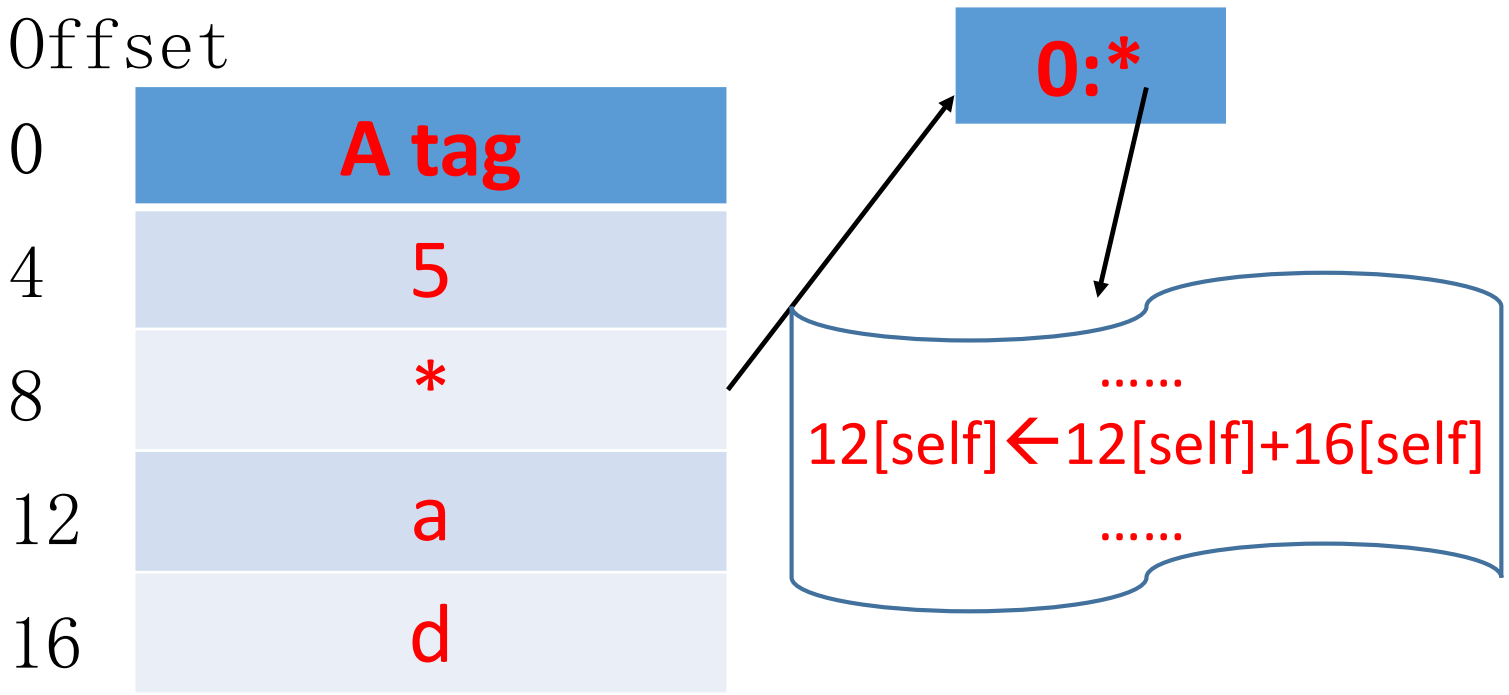
- The first 3 words of Cool objects contain header information:

		Offset
Identifies class of the object	**Class Tag**	0
Size of the object in words	Object size	4
a pointer to a table of methods	Dispatch Pointer	8
	Attribute 1	12
	Attribute 2	16
	…	
	…	
	…	

Lay out in contiguous memory

Cool Object Layout (Cont.)

```
Class A {
        a: Int <- 0;
        d: Int <- 1;
        f(): Int { a <- a + d };
        };
```



Lay out in contiguous memory

Inheritance

```
Class A {
        a: Int <- 0;
        d: Int <- 1;
        f(): Int { a <- a + d };
        };
Class B inherits A {
        b: Int <- 2;
        f(): Int { a }; // Override
        g(): Int { a <- a - b };
};
```

```
Class C inherits A {
        c: Int <- 3;
        h(): Int { a <- a * c };
};
```

- Attributes a and d are inherited by classes B and C
- All classes refer to a
- For f method to work correctly in A, B, and C objects, attribute a must be in the same "place" in each object

Object Layout ?

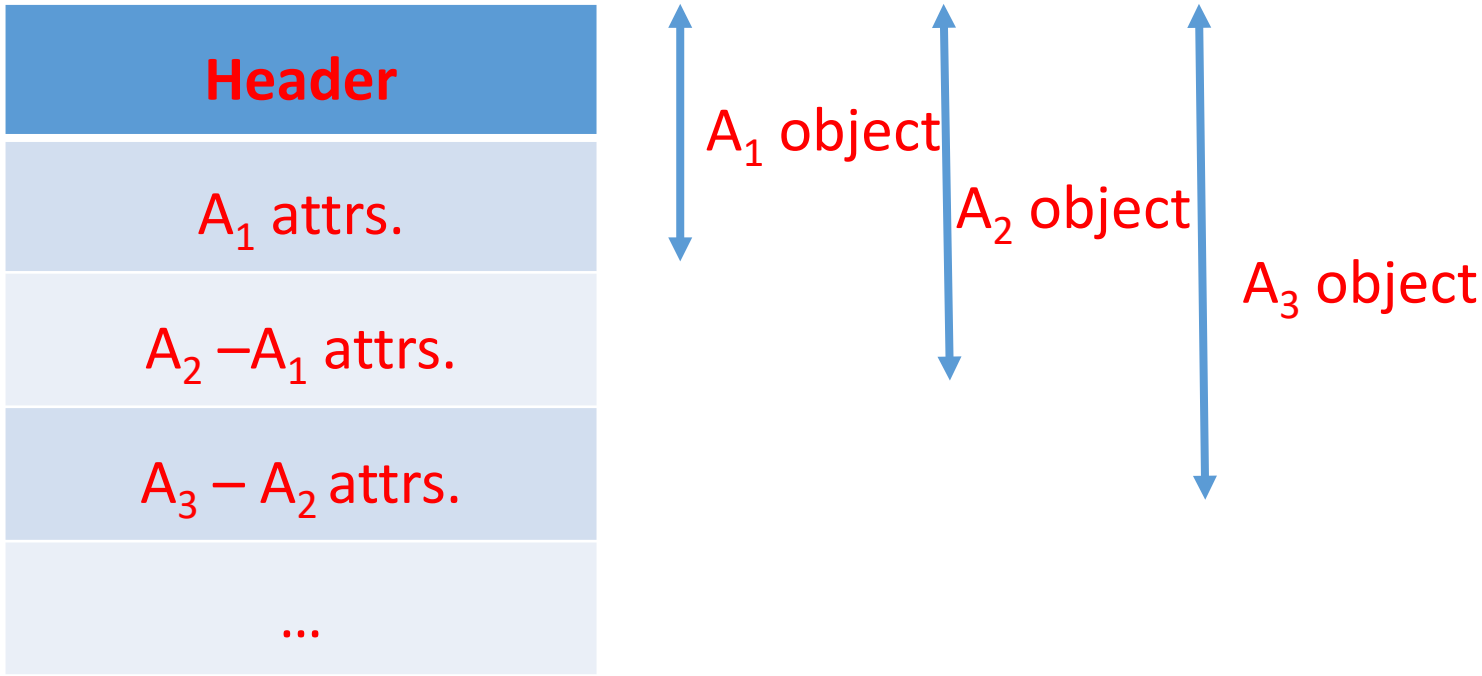
Subclasses

- Observation: Given a layout for class A, a layout for subclass B can be defined by extending the layout of A with additional slots for the additional attributes of B
- Leaves the layout of A unchanged (B is an extension)

Class Offset	A	B	C	
0	A tag	B tag	C tag	Class tag
4	5	6	6	Object size
8	*	*	*	Dispatch Pointer
12	a	a	a	Attribute
16	d	d	d	Attribute
20		**b**	**c**	Attribute

Subclasses (Cont.)

- The offset for an attribute is the same in a class and all of its subclasses
  - Any method for an A_1 can be used on a subclass A_2
- Consider layout for A_n inherits from A_{n-1} inherits from … A_2 inherits from A_1



Dynamic Dispatch

```
Class A {
        a: Int <- 0;
        d: Int <- 1;
        f(): Int { a <- a + d };
        };
Class B inherits A {
        b: Int <- 2;
        f(): Int { a }; // Override
        g(): Int { a <- a - b };
};
```

```
Class C inherits A {
        c: Int <- 3;
        h(): Int { a <- a * c };
};
```

- **e.g()**
  - **g refers to method in B if e is an instance of B**
- **e.f()**
  - **f refers to method in A if e is an instance of A or C**
  - **f refers to method in B if e is an instance of B**

Dispatch Tables

- Every class has a fixed set of methods (including inherited methods)
- A dispatch table indexes these methods
  - An array of method entry points
  - A method f lives at a fixed offset in the dispatch table for a class **and all of its subclasses**

Class Offset	A	B	C	
0	A.f	B.f	A.f	Pointer to method entry of f
4		g	h	Pointer to method entry

- The tables for B and C extend the table for A with more methods
- Because methods can be overridden, the method for f is not the same in every class, but is always at the same offset

Using Dispatch Tables

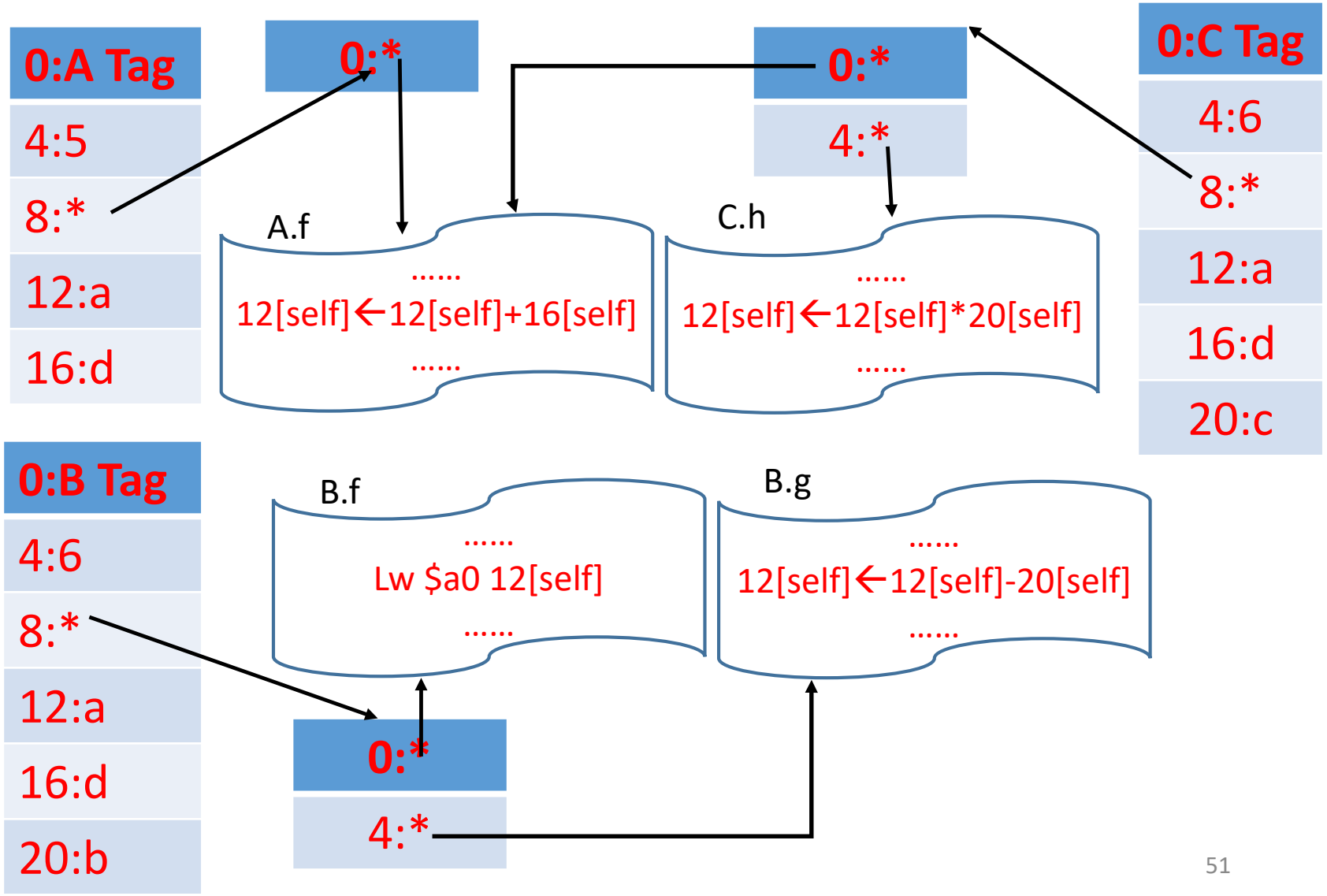
- The dispatch pointer in an object of class X points to the dispatch table for class X
- Every method f of class X is assigned an offset O_f in the dispatch table at compile time
- Every method must know what object is "self"
  - "self" is passed as the first argument to all methods
- To implement a dynamic dispatch e.f() we
  - Evaluate e, obtaining an object x
  - Find the dispatch table D by reading the dispatch-pointer field of x
  - Call $D[O_f](x)$

    In the call, self is bound to x

Using Dispatch Tables

```
Class A {
        a: Int <- 0;
        d: Int <- 1;
        f(): Int { a <- a + d };
        };
Class B inherits A {
        b: Int <- 2;
        f(): Int { a }; // Override
        g(): Int { a <- a - b };
};

Class C inherits A {
        c: Int <- 3;
        h(): Int { a <- a * c };
};
```



Multiple Inheritance

- Assume that we extend Cool with multiple inheritance
- Consider the following 3 classes:

```
Class A { a : Int; m1() : Int { a }; }
Class B { b: Int; m2() : Int { b }; }
Class C inherit A, B { c : Int; m1() : Int { c }; }
```

- class C inherits attribute a and overrides method m1 from A, inherits attribute b and method m2 from B

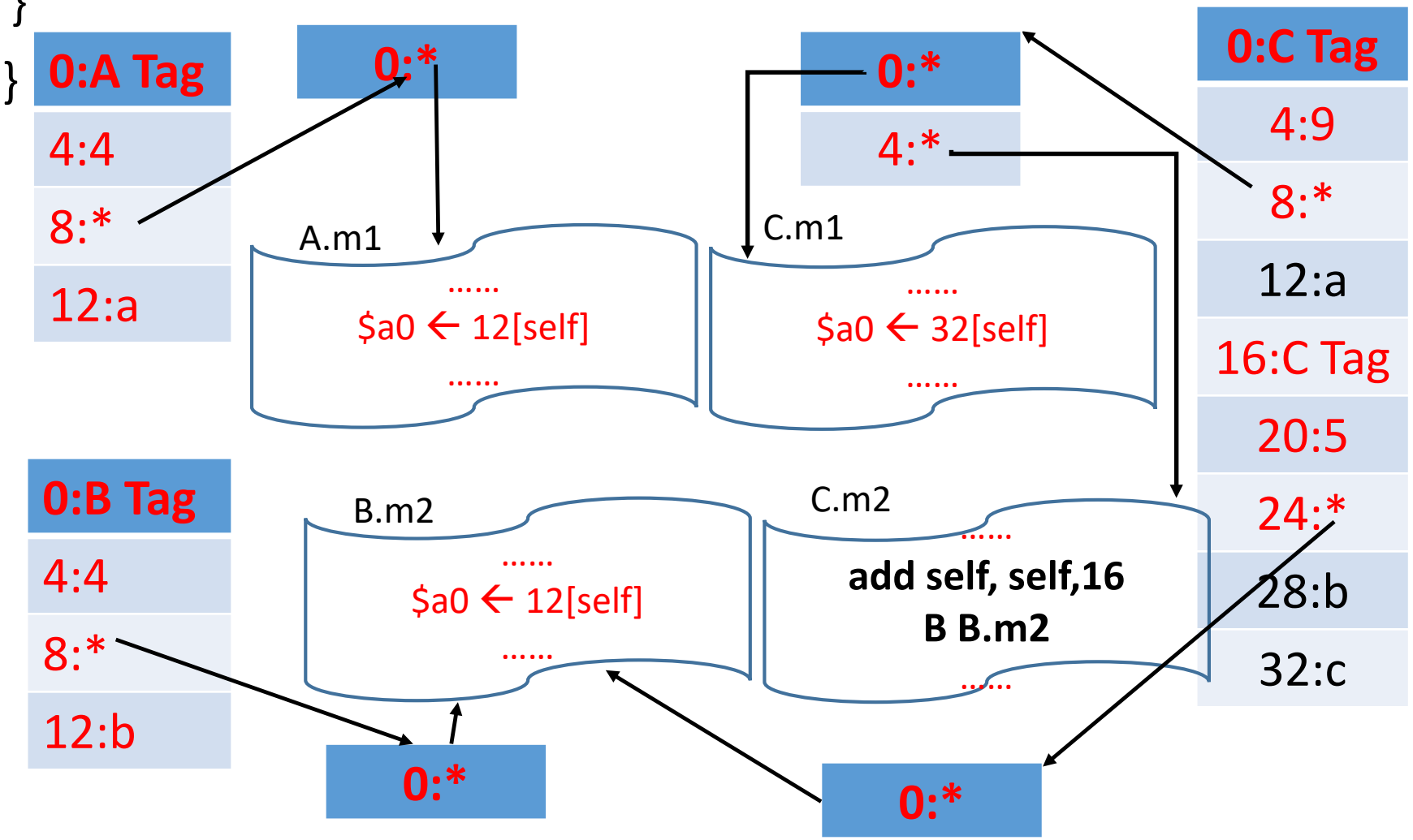
Object Layout

Class A { a : Int; m1() : Int { a }; }

Class B { b: Int; m2() : Int { b }; }

Class C inherit A, B

{ c : Int; m1() : Int { c }; }



Refer to C3 algorithm for more details

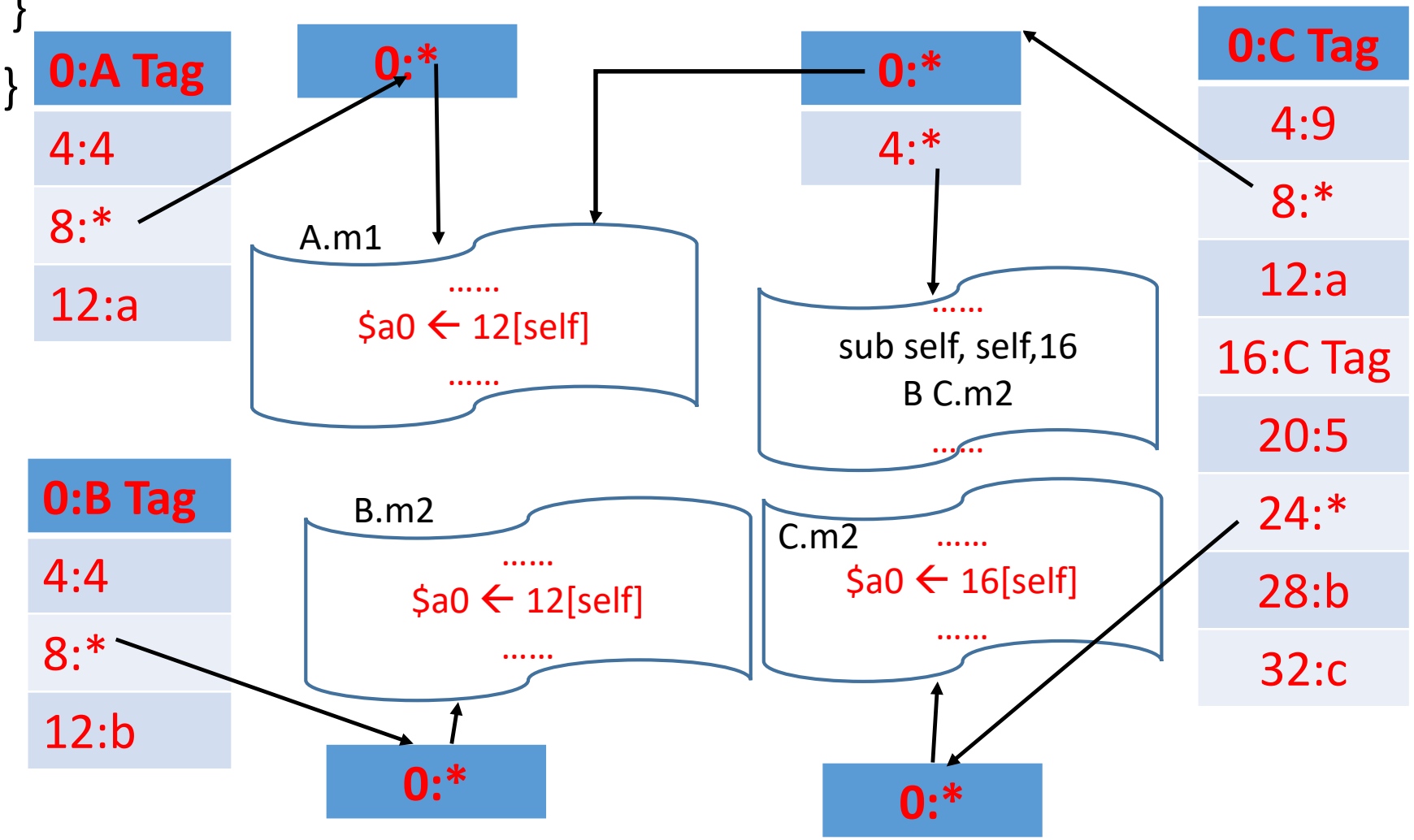
Object Layout

Class A { a : Int; m1() : Int { a }; }

Class B { b: Int; m2() : Int { b }; }

Class C inherit A, B

{ c : Int; m2() : Int { c }; }



Compilation Schemes

- AST to MIPS programs (For PA4, used by JVM)
  - AST to Stack machines
  - Stack machines to MIPS assembly language (RISC)
  - Object Oriented Code Generation
    - ✓ Object Layout, Dynamic Dispatch
- Three-address to a Simple Target-Machine （CISC-like, from textbook）
- Both of them can be generalized to other target-machines and IRs

The Target Machine

- Familiarity with instruction set of target is a prerequisite for good code generation
- Textbook uses a representative assembly language of a representative target computer
  - Byte-addressable machine with four bytes to a word
  - Machine uses `n` general-purpose registers `R0`, `R1`, ... `Rn-1`
  - Three-address instructions of form:
    - **`op destination,source1,source2`**
  - The op-codes include `MOV`, `ADD`, and `SUB` (among others)
  - Certain bit patterns in source and destination fields specify that words following instruction contain operands and/or addresses
- Techniques discussed are more general and have been used on several classes of machines

Address Modes of Sample Machine

MODE	FORM	ADDRESS	ADDED COST	Example
Absolute	M	M	1	MOV R1, 0x1003
register	R	R	0	MOV R1, R2
indirect register	[R]	contents(R)	0	MOV R1,[R2]
indexed	[R,#c]	contents(R)+ c	1	MOV R1,[R2,#0x1003]
indirect indexed	*c(R)	contents(c + contents(R))	1	MOV R1,*0x1003(R2)
MODE	**FORM**	**CONSTANT**	**ADDED COST**	
literal	#c	c	1	MOV R1, #0x1003

Instruction Costs

- Cost of one instruction for sample computer is:
  - Equal to one + the costs associated with the source and destination address modes
  - Corresponds to the length (in words) of the instruction
  - Address modes involving registers have cost zero
  - Those with memory location or literal have cost one since operands have to be stored with instruction
- Time taken to fetch instruction often exceeds time spent executing instruction

Example

a := b + c

- The following solution has cost 6:

```
MOV R0, b
ADD R0,R0,c
MOV a, R0
```

- The following solution also has cost 6:

```
MOV a, b
ADD a,a,c
```

- If `R0`, `R1`, and `R2` contain the addresses of `a`, `b`, and `c`, respectively, the cost can be reduced to 2:

```
MOV [R0],[R1]
ADD [R0],[R0],[R2]
```

- If `R1` and `R2` contain the values of `b` and `c`, respectively, and `b` is not needed again, the cost can be reduced to 3:

```
ADD R1,R2,R1
MOV a, R1
```

Run-time storage management

- Activation records store information needed during the execution of a procedure
- Two possible storage-allocation strategies for activation records are:
  - static allocation // static
  - stack allocation // dynamic
- An activation record has fields which hold:
  - result and parameters
  - machine-status information
  - local data and temporaries
- Size and layout of activation records are communicated to code generator via information in the symbol table

Static Allocation

- A `call` statement in the intermediate code is implemented by two target-machine instructions:

```
MOV callee.static_area, #here+20
JMP callee.code_area
```

- The `MOV` instruction saves the return address
- The `JMP` statement transfers control to the target code of the called procedure
- A return statement in the intermediate code is implemented by one target-machine instruction:

```
JMP *callee.static_area
```

Sample Code

```
// procedure c
    action1
    Call p
    action2
    HALT

//procedure p
    Action3
    return
```

```
                          /* code for c */
100:  action1
120:  MOV 364, #140  /* save return address 140=120+20 */
132:  JMP 200       /* call p */
140:  action2
160:  HALT
      …
                          /* code for p */
200:  action3
220:  JMP *364
      …
                          /* 300-363: activation record for c */
300:                      /* return address */
304:                      /* local data for c */
      …
                          /* 364-451: activation record for p */
364: 140                  /* return address */
368:                      /* local data for p */
```

Stack Allocation

- Stack allocation uses relative addresses for storage in activation records
  - Address can be offset from any known position in activation record
  - Usually uses positive offset from `sp`, a pointer to beginning of activation record at top of stack
- The position of an activation record is not known until run-time
  - This position is usually stored in a register `sp`
  - The indexed address mode is convenient, e.g., [sp, #offset], *offset(sp)

Stack Allocation (cont.)

- The code for the first procedure initializes the stack:

```
MOV SP, #stackstart
… code for the first procedure …
HALT
```

- A procedure call increments `SP`, saves the return address, and transfers control:

```
ADD SP,SP, #caller.recordsize  [stack from
                                 low to high]
MOV SP, #here+16
JMP callee.code_area
```

- A return sequence has two parts:
  - First control is transferred to the return address

```
JMP [SP,#0]
```

  - Next, in the caller, SP is restored to its previous value

```
SUB SP,SP, #caller.recordsize
```

Basic Blocks

- A basic block is a sequence of statements such that:
  - Flow of control enters at start
  - Flow of control leaves at end
  - No possibility of halting or branching except at end
- Each basic block has a first statement known as the "leader" of the basic block
- A name is "live" at a given point if its value will be used again in the program

Partitioning Code into Basic Blocks

- First algorithm must determine all leaders:
  - The first statement is a leader
  - Any statement that is the target of a conditional or unconditional jump/call is a leader
  - Any statement immediately following a conditional or unconditional jump is a leader
  - These are all used to Control Flow Integrity(CFI) checking for preventing code-reuse attacks
- A basic block (assume no indirect jump):
  - Starts with a leader
  - Includes all statements up to but not including the next leader

Basic Block Example

```
begin
  prod := 0;
  i := 1;
  do
    begin
      prod := prod + a[i] * b[i]
      i := i+1
    end
  while i <= 20
end
```

⟹

```
 (1) prod := 0
 (2) i := 1
 (3) t1 := 4 * i
 (4) t2 := a[t1]
 (5) t3 := 4 * i
 (6) t4 := b[t3]
 (7) t5 := t2 * t4
 (8) t6 := prod + t5
 (9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 jmp 3
(13) …
```

Next-Use Information

- The use of a name in a three-address statement is defined as follows:
  - Suppose statement i assigns a value to `x (i.e., x:=e)`
  - Suppose statement `j` has `x` as an operand
  - We say that `j` uses the value of `x` computed at `i` if:
    - There is a path through which control can flow from statement i to statement j (taint analysis)
    - This path has no intervening assignments to x

```
1: x:= 2;
2: x:= x*x;
3: y:= x-1;
```

Next-Use Information

- For each three-address statement that is of the form `x:=y op z`
  - We want to determine the next uses of `x`, `y`, and `z`
  - For now, we do not consider next uses outside of the current basic block

Determine Next-Use

- Scan each basic block from end to beginning
  - At start, record, if known, which names are live on exit from that block
  - Otherwise, assume all non-temporaries are live
  - If algorithm allows certain temporaries to be live on exit from block, consider them live as well
- Whenever reaching a three address statement at line `i` with the form `x := y op z`
  - Attach statement `i` to information in symbol table regarding the next use and liveness of `x`, `y`, and `z`
  - Next, in symbol table, set `x` to "not live/killed" and "no next use"
  - Then set `y` and `z` to "live" and the next uses of `y` and `z` to `i`
- A similar approach is taken for unary operators

A Simple Code Generator

- Generates target code for a sequence of three-address statements
- Considers statements one at a time:
  - Remembers if any of the operands are currently in registers
  - Takes advantage of that if possible
- Assumes that for each operator in three-address code there is a corresponding target-language operator
- Also assumes that computed results can be left in registers as long as possible, storing them only:
  - If their register is needed for another computation or end of the block

Register and Address Descriptors

- A register descriptor keeps track of what is currently in each register
  - Consulted whenever a new register is needed
  - Assumes that all registers are empty at the start of a basic block

- An address descriptor keeps track of the location where the current value of a name can be found
  - The location might be a register, a memory address, a stack location, or some set of these
  - The information can be stored in a symbol table
  - The information is used to determine how to access the value of a name

Code-Generation Algorithm

- For each three-address statement `x := y op z` perform the following actions:
  - Invoke a function **`getreg`** which determines the location `L` where the result should be stored
  - `L` will usually be a register but could be a memory location
  - Consult the address descriptor for `y` to determine L_y, one of the current locations of `y`
  - If the value of y is not already in L, generate the instruction `MOV L,` L_y to place a copy of `y` in `L`
  - Generate the instruction `OP L, L,` L_z, where L_z is the chosen current location of `z`
  - If the current value of y or z is in a register and is no longer needed, update the register descriptor (next use information)
- The actions for unary operators are analogous
- A simple assignment statement is a special case

Implementing `getreg`

- The function returns the location `L` to hold the result of `x:= y op z`
- If `y` is already in a register R:
  - `L` can be set to `R` if the following conditions are true:
    - The name `y` is not live after exiting the block
    - There is no next use of `y` after that, then return R.
  - In this case, update the address descriptor of `y` to indicate that the value of `y` is no longer in `L`.
- Otherwise, if there is an empty register, use it
- Otherwise, if `x` has a next use in the block or `op` requires a register, then:
  - Choose an occupied register `R`
  - For every variable in `R`
    - If the variable is not also in memory, execute an instruction `MOV M, R`
    - Update the appropriate address descriptor
  - return `R`
- Otherwise, select the memory location of `x` for `L`

Code Generation Example (1)

- Consider the statement

```
d := (a - b) + (a - c) + (a - c)
```

- This may be translated into the following three-address code:

```
t := a - b
u := a - c
v := t + u
d := v + u
```

- Assume that `d` is live at end of block
- Assume that `a`, `b`, and `c` are always in memory
- Assume that `t`, `u`, and `v`, being temporaries, are not in memory unless explicitly stored with a `MOV` instruction

Code Generation Example (2)

Statements	Generated Code	Register Descriptor	Address Descriptor
		empty	empty
t := a - b	MOV R0, a SUB R0,R0,b	R0 contains t	t in R0

```
t := a - b
u := a - c
v := t + u
d := v + u
```

getreg(a)=R0

Code Generation Example (2)

Statements	Generated Code	Register Descriptor	Address Descriptor
		empty	empty
t := a - b	MOV R0, a SUB R0,R0,b	R0 contains t	t in R0
u := a - c	MOV R1, a SUB R1,R1,c	R0 contains t R1 contains u	t in R0 u in R1

```
t := a - b
u := a - c
v := t + u
d := v + u
```

getreg(a)=R1

Code Generation Example (2)

Statements	Generated Code	Register Descriptor	Address Descriptor
		empty	empty
t := a - b	MOV R0, a SUB R0,R0,b	R0 contains t	t in R0
u := a - c	MOV R1, a SUB R1,R1,c	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R0,R0,R1	R0 contains v R1 contains u	v in R0 u in R1

```
t := a - b
u := a - c
v := t + u
d := v + u
```

```
getreg(t)=R0
as t is not live
```

Code Generation Example (2)

```
t := a - b
u := a - c
v := t + u
d := v + u
```

Statements	Generated Code	Register Descriptor	Address Descriptor
		empty	empty
t := a - b	MOV R0, a SUB R0,R0,b	R0 contains t	t in R0
u := a - c	MOV R1, a SUB R1,R1,c	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R0,R0,R1	R0 contains v R1 contains u	v in R0 u in R1
d := v + u	ADD R0,R0,R1	R0 contains d	d in R0
	MOV d, R0	R0 contains d	d in R0 and memory

getreg(v)=R0

As v is not live

also u is not live

Update register and address descriptor

Conditional Statements (1)

- Implemented in one of two ways
- One way: branch if value of designated register meets one of six conditions
  - Six conditions are negative, zero, positive, nonnegative, nonzero, and nonpositive
  - To implement a three-address statement such as

    `if x < y goto z`

    - Subtract `x` from `y` in register `R (same as above)`
    - If value in `R` is negative jump to `z`

```
MOV x, R0
SUB y, R0
CJ< z
```

Conditional Statements (2)

- A second approach uses a set of condition codes
- Codes indicate whether the last quantity computed or loaded into a register is negative, zero, or positive (RFLAGS register)
- Many machines use a compare instruction `cmp` that sets a condition code without subtracting
- Other instructions will compare and jump if the condition code meets certain criteria
- To implement the statement `if x < y goto z`, first execute `CMP x, y` and then `CJ< z`
- A condition-code descriptor stores the name (or pair of names compared) that last set the condition code

```
x := y + z
if x < 0 goto z
```

⟹

```
MOV R0, y
ADD R0,R0,z,
MOV x, R0
CJ< z
```

Register Allocation

- Goal: assign more temporaries to a register, but without changing the program behavior (improve performance: time、size、memory)
- Register allocation decides which values in a program reside in registers
- Register assignment decides in which register each value resides
- Often certain registers are reserved for certain purposes:
  - base register
  - stack pointer
  - frame pointer
- Remaining registers remain free for compiler to do with what it likes

An Example

- Consider the program

  a := c + d

  e := a + b

  f := e - 1

  – with the assumption that a and e die after use
- Temporary a can be “reused” after “a + b”
- Same with temporary e after “e - 1”
- Can allocate a, e, and f all to one register (r1):

```
r1 := c + d
r1 := r1 + b
r1 := r1 - 1
```

Global Register Allocation

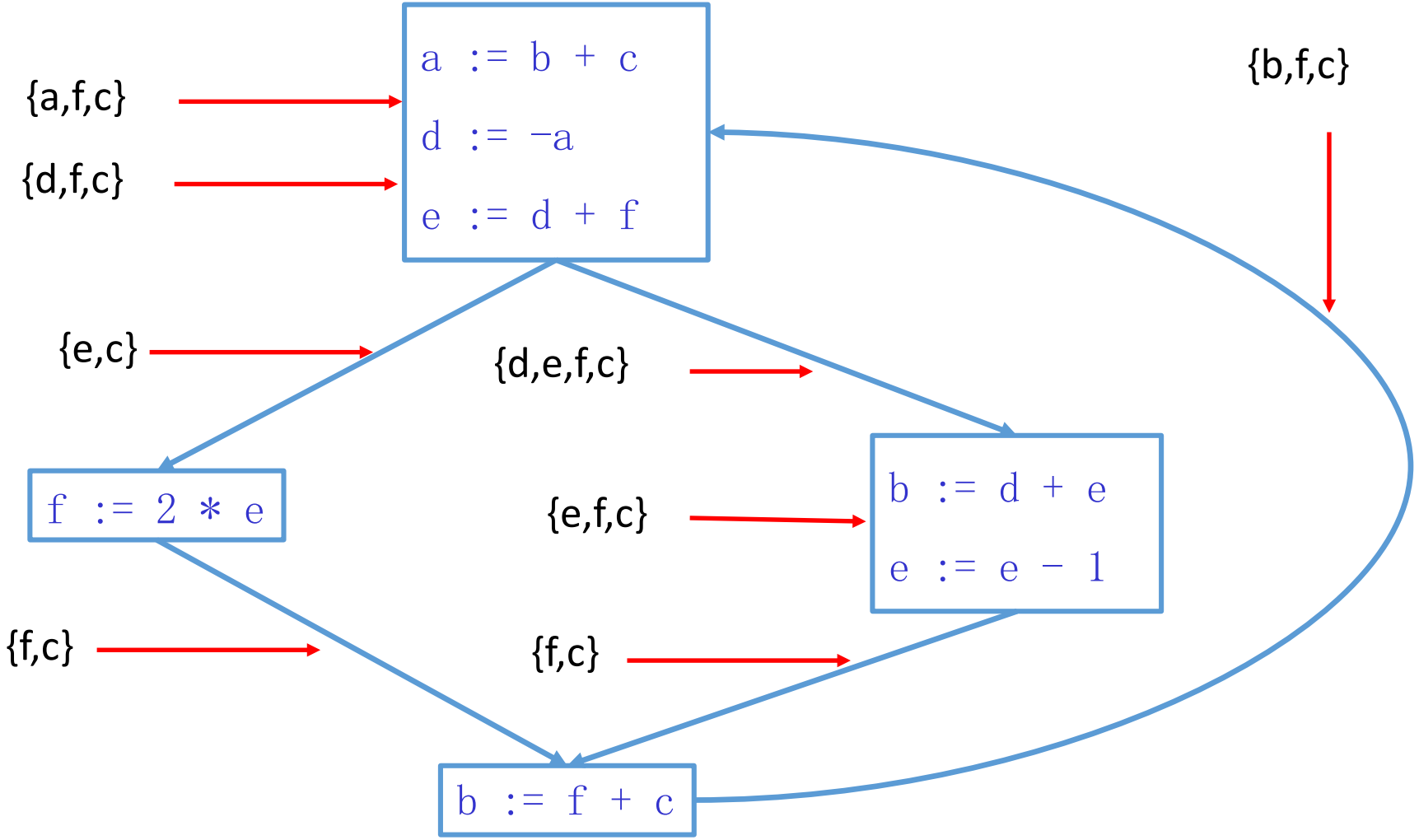
- Previously discussed strategy stores all registers at the end of each block
- To eliminate stores and loads, assign some registers to frequently used variables
  - Keep these registers consistent across block boundaries
  - Assuming we know which values are used outside of a basic block
  - Fact: many programs spend most of their execution time in inner loops
  - Some compilers assign fixed number of registers to hold the most active values in each inner loop
- Some languages allow programmer to specify register declarations

Basic Register Allocation Idea

- The value in a dead temporary is not needed for the rest of the computation
  - A dead temporary can be reused
- Basic rule:

  Temporaries t_1 and t_2 can share the same register if at any point in the program at most one of t_1 or t_2 is live !

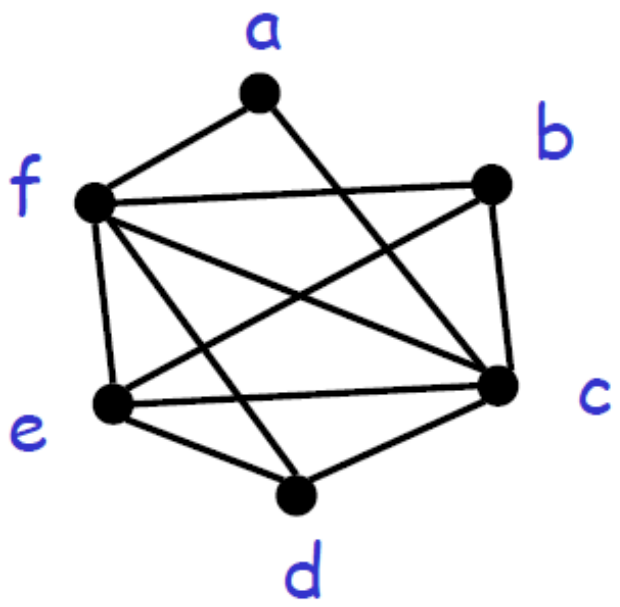
Global live variables

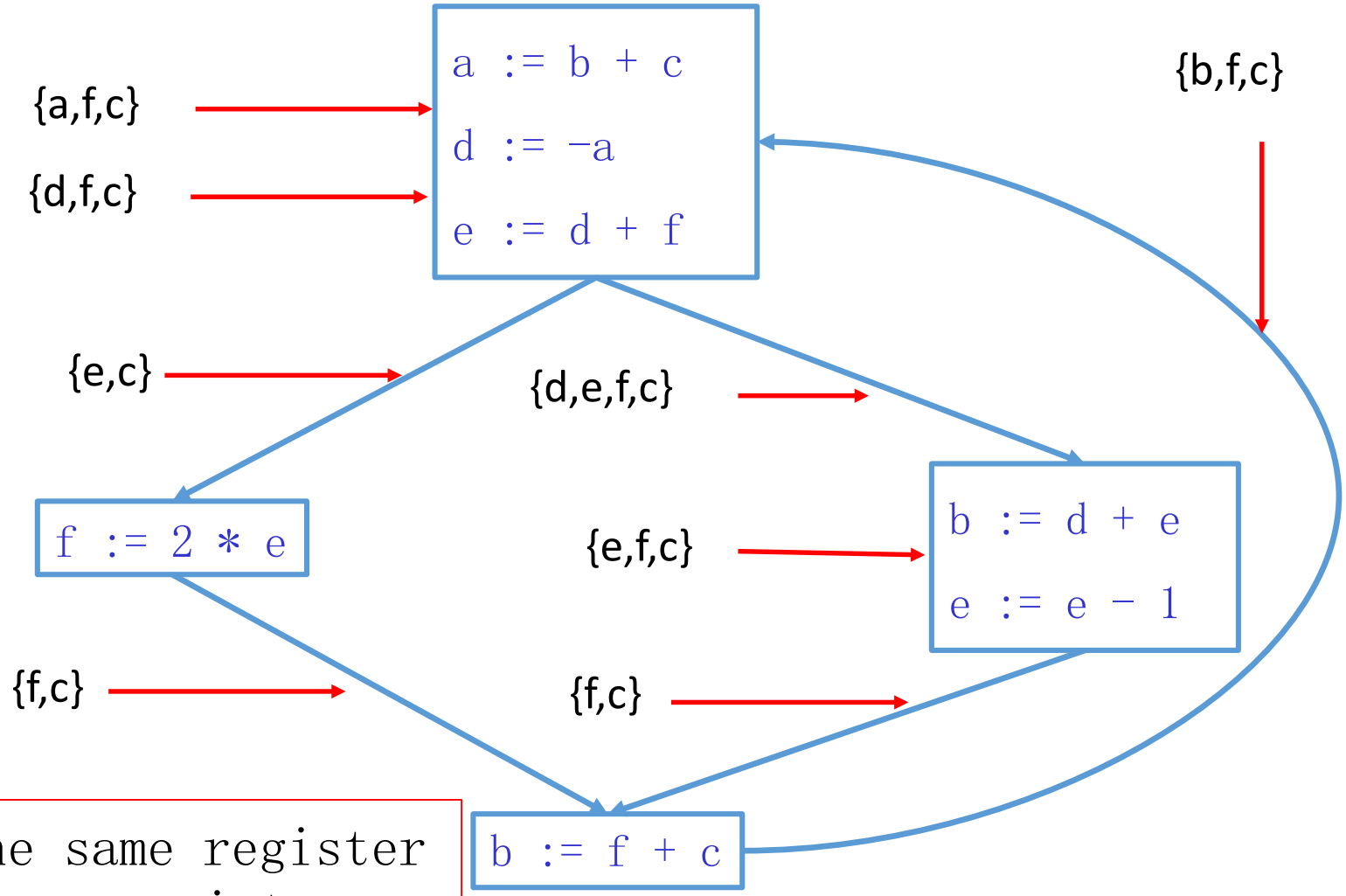


The Register Interference Graph

- Two temporaries that are live simultaneously cannot be allocated in the same register
- We construct an undirected graph
  - A node for each temporary
  - An edge between t1 and t2 if they are live simultaneously at some point in the program
- This is the register interference graph (RIG)
  - Two temporaries can be allocated to the same register if there is no edge connecting them

Global live variables





E.g., b and c cannot be in the same register
E.g., b and d can be in the same register

Register Interference Graph Properties

- It extracts exactly the information needed to characterize legal register assignments
- It gives a global (i.e., over the entire flow graph) picture of the register requirements
- After RIG construction the register allocation algorithm is architecture independent

Graph Coloring

A coloring of a graph is an assignment of colors to nodes, such that nodes connected by an edge have different colors

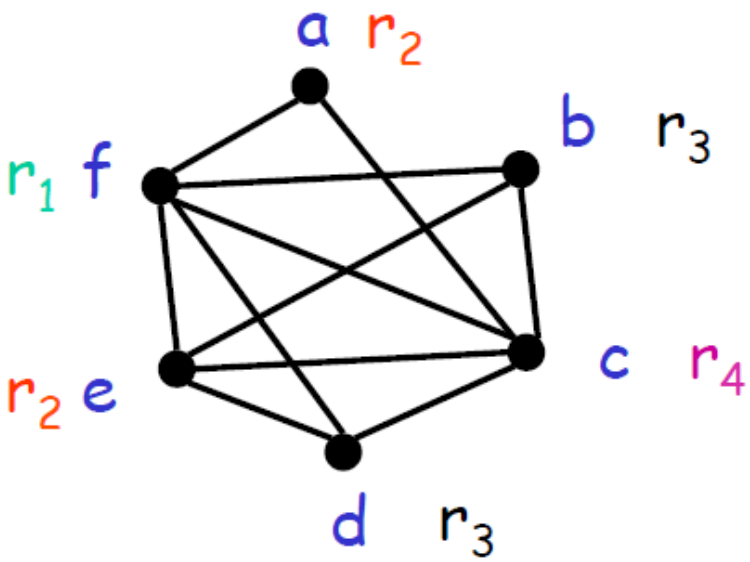
A graph is k-colorable if it has a coloring with k colors

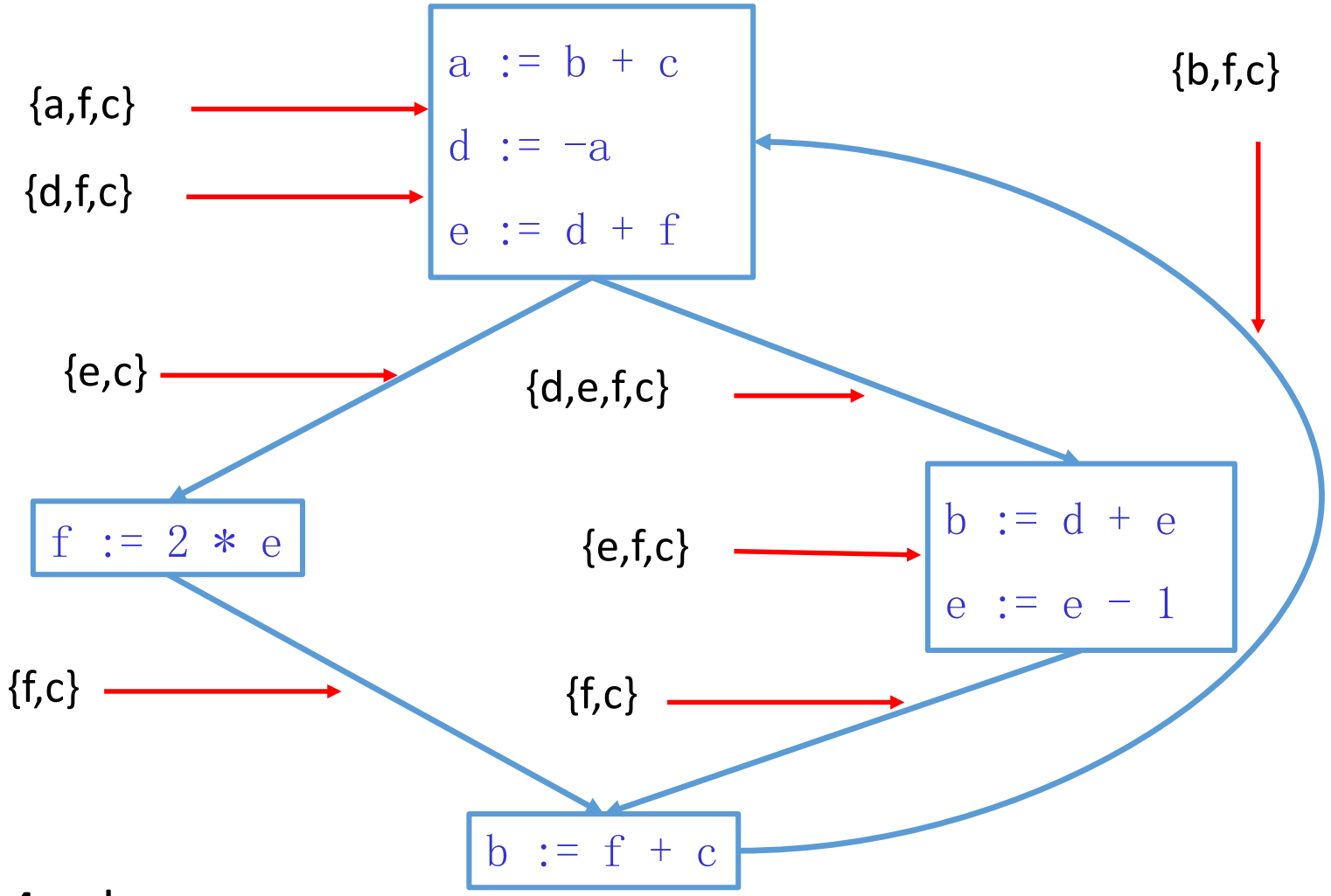
colors = registers

– We need to assign colors (registers) to graph nodes (temporaries)

- Let k = number of machine registers
- If the RIG is k-colorable then there is a register assignment that uses no more than k registers

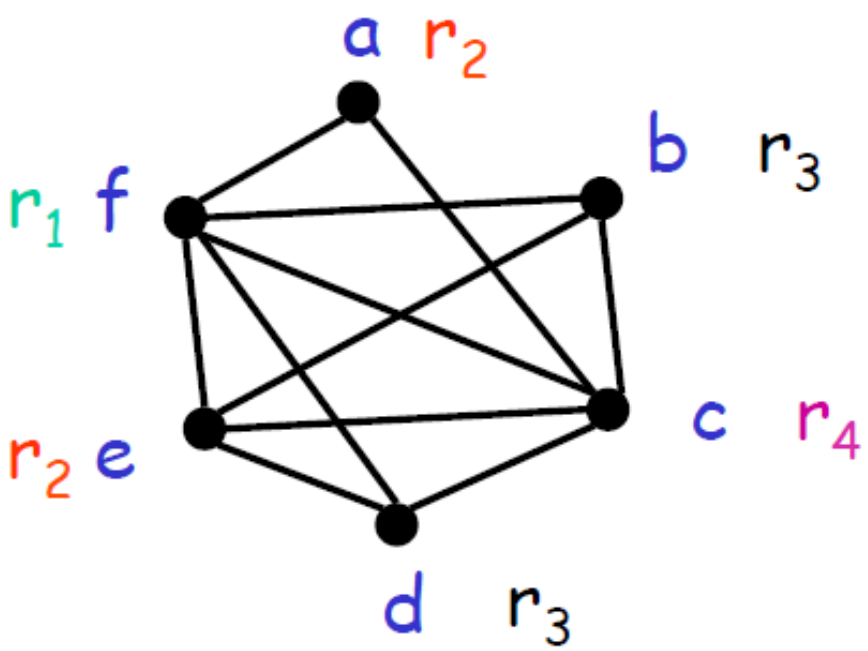
Register allocation

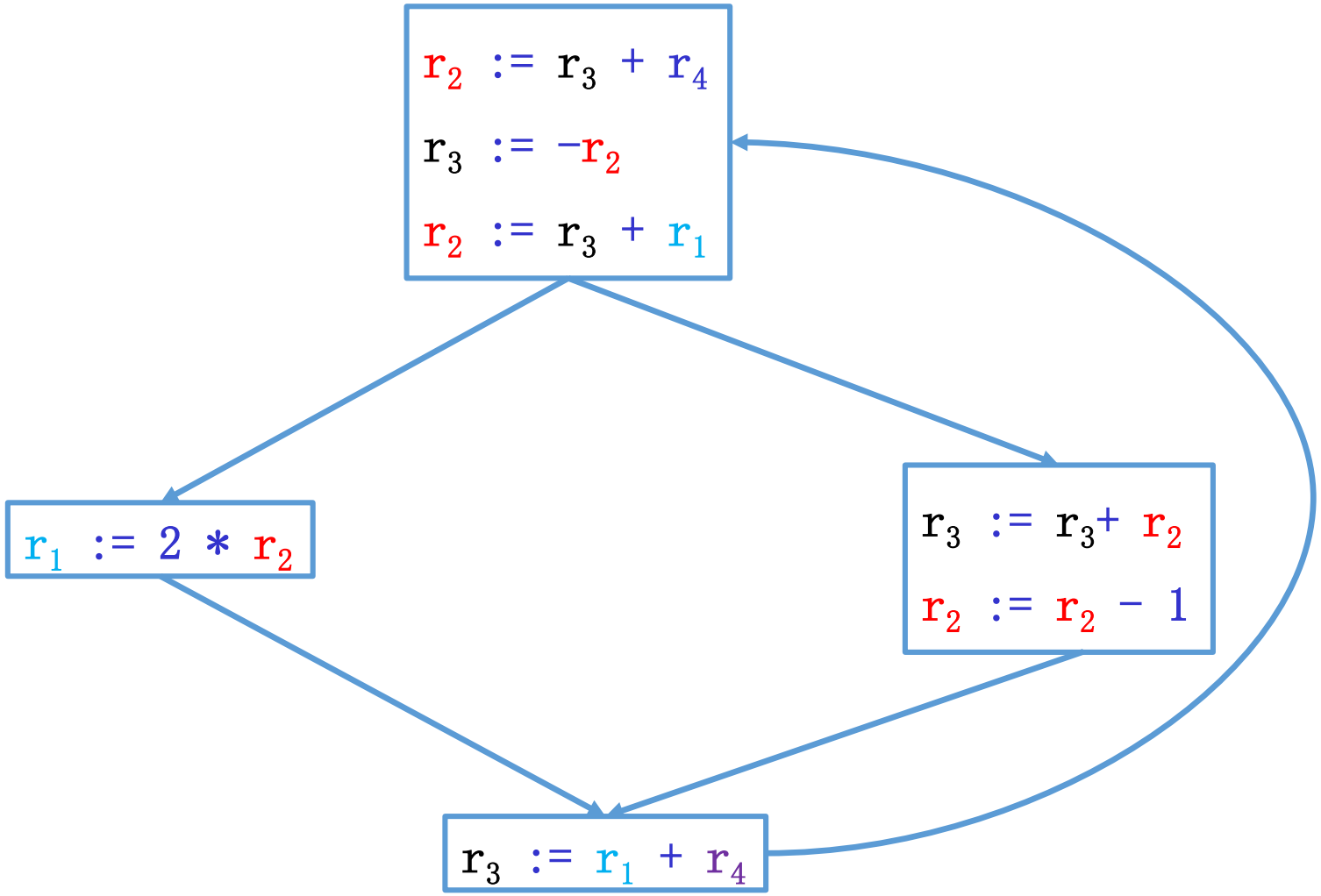




- There are 4-colorings of this graph
- There is no coloring with less than 4 colors

Register allocation





Register allocation

The remaining problem is to compute a coloring for the interference graph

- But:
  1. This problem is very hard (NP-hard).
  2. No efficient algorithms are known.
  3. A coloring might not exist for a given number or registers
- The solution to (1) is to use heuristics

Observation:

- Pick a node t with fewer than k neighbors in RIG
- Eliminate t and its edges from RIG
- Soundness: If the resulting graph has a k-coloring then so does the original graph

- Why:
  - Let $c_1,\ldots,c_n$ be the colors assigned to the neighbors of t in the reduced graph
  - Since $n < k$ we can pick some color for t that is different from those of its neighbors

Register allocation

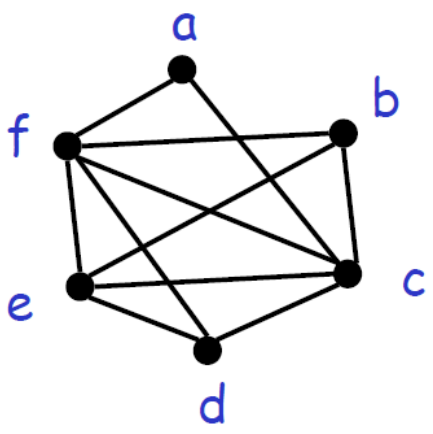
- The following works well in practice:
  - Pick a node t with fewer than k neighbors
  - Push t on a stack and remove it from the RIG
  - Repeat until the graph has one node
- Then start assigning colors to nodes in the stack (starting with the last node added)
  - At each step pick a color different from those assigned to already colored neighbors

Register allocation

Start with the RIG and with k = 4

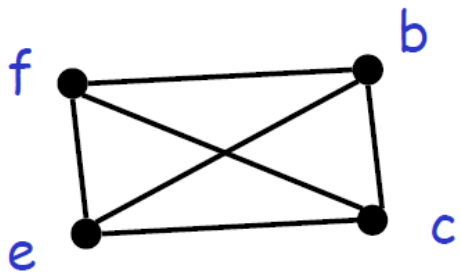
Stack= [e,b,c,d,a]

Stack= []



Remove a and then d

Stack= [d,a]



Then removed c, b, e

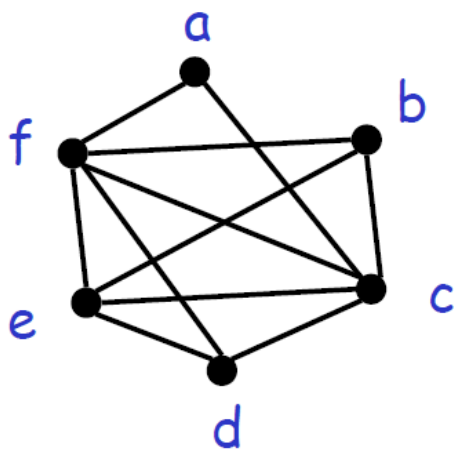
f ●

$f=r_1$	available $\{r_1,r_2,r_3,r_4\}$
$e=r_2$	available $\{r_2,r_3,r_4\}$
$b=r_3$	available $\{r_3,r_4\}$
$c=r_4$	available $\{r_4\}$
$d=r_3$	available $\{r_3\}$
$a=r_2$	available $\{r_2,r_3\}$

At each step pick a color different from those assigned to already colored neighbors

What if the Heuristic Fails?

Example: try to find a 3-coloring of the RIG



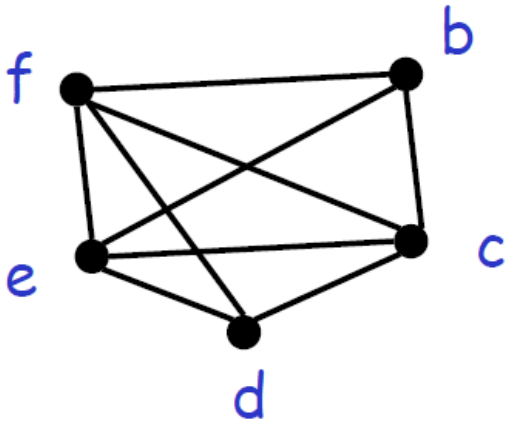
What happen?

What if during simplification we get to a state where all nodes have k or more neighbors ?

Remove a and get stuck

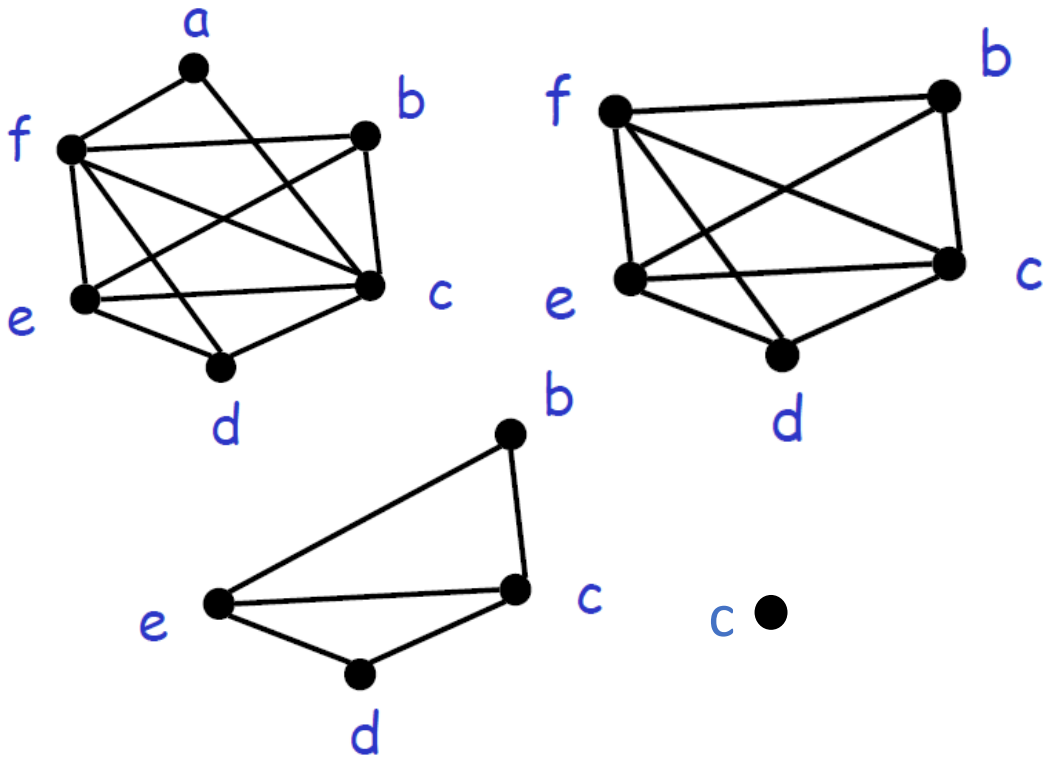
Pick a node as a candidate for spilling

– A spilled temporary "lives" in memory



What if the Heuristic Fails?

Example: try to find a 3-coloring of the RIG



1. Remove a and get stuck
2. f is picked as a candidate for spilling
3. Now, we can remove b, d, e

$c=r_1$	available $\{r_1,r_2,r_3\}$
$e=r_2$	available $\{r_2,r_3\}$
$d=r_3$	available $\{r_3\}$
$b=r_3$	available $\{r_3\}$
f=?	available { }

We must allocate a memory location as the home of f

- Typically this is in the current stack frame
- Call this address fa

- Before each operation that uses f, insert f := ld fa
- After each operation that defines f, insert st f, fa

Register Allocation

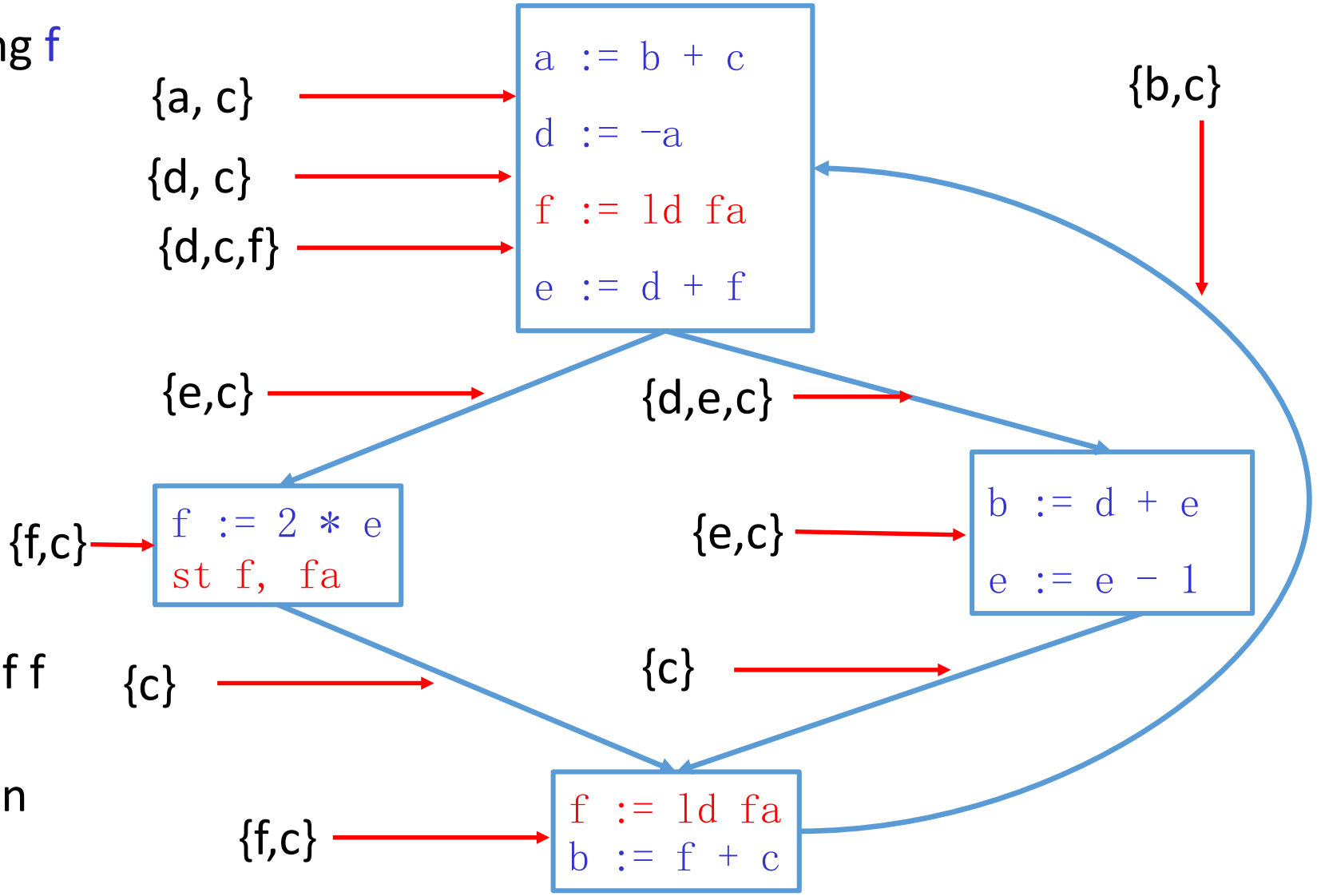
This is the new code after spilling f

Liveness information after spilling should be recomputed

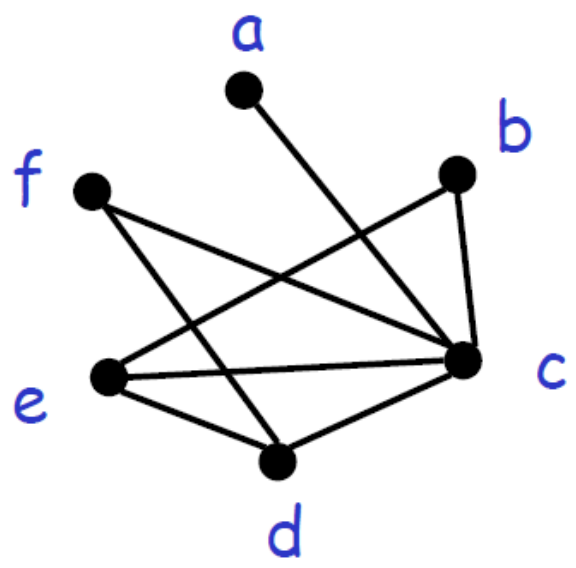
f is live only
- just after f := ld fa
- just before st f, fa

Spilling reduces the live range of f

thus result in fewer neighbors in RIG for f



Register Allocation



try to find a 3-coloring of the RIG

Register Allocation

- Additional spills might be required before a coloring is found
- The tricky part is deciding what to spill
- Possible heuristics:
  - Spill temporaries with most conflicts
  - Spill temporaries with few definitions and uses
  - Avoid spilling in inner loops
- Any heuristic is correct

Exercises

```
int i=1,j=2,a,b=2,c=3;
if (i<j)
   a=b+c;
else
   a=b-c;
```

Generate its assembly code