

Compilers

Design and Implementation

Register Allocation

Introduction

Local Register Allocators

Copyright 2023, Pedro C. Diniz, all rights reserved.

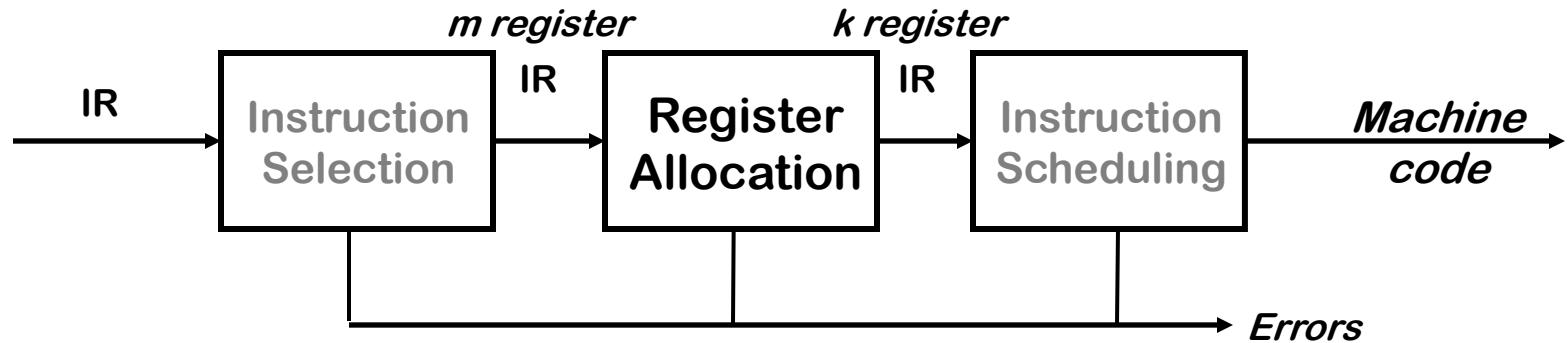
Students enrolled in the Compilers class at Faculdade de Engenharia da Universidade do Porto (FEUP) have explicit permission to make copies of these materials for their personal use.

Outline

- What is Register Allocation and Its Importance
- Simple Register Allocators
- Webs
- Interference Graphs
- Graph Coloring
- Splitting
- More Transformations

What is Register Allocation?

Part of the Compiler's Back End



Critical Properties

- Produce Correct Code that Uses k (or fewer) Registers
- Minimize Added Loads and Stores
- Minimize Space Used to Hold *Spilled Values*
- Operate Efficiently
 - $O(n)$, $O(n \log n)$, maybe $O(n^2)$, but not $O(2^n)$

Register Allocation & Assignment



Optimal global allocation
is NP-Complete, under
almost any assumptions.

- At Each Point in the Code
 1. Allocation: Determine which Values will reside in Registers
 2. Assignment: Select a Register for each such Value

The Goal:

Allocation that “Minimizes” Running Time

Importance of Register Allocation

- *Optimally* Use of one of the Most Critical Processor Resources
 - Affects almost every Statement of the Program
 - Register Accesses are much faster than Memory Accesses
 - Eliminates expensive memory instructions
 - Wider gap in faster newer processors
 - Number of instructions goes down due to direct manipulation of registers (no need for load and store instructions)
- *Probably* the optimization with the most impact!
- Common Trade-Off:
 - Registers: Fast Storage with Small Capacity (say 32, 64, 128)
 - Main Memory: Slow Storage with High Capacity (say Giga Bytes)

Importance of Register Allocation

- What Can Be Put in Registers?
 - Scalar Variables
 - Big Constants
 - Some Array Elements and Record Fields
 - Register set depending on the data-type
 - Floating-point in FP registers
 - Fixed-point in Integer registers
- Allocation of Variables (including temporaries) up-to-now stored in Memory to Hardware Registers
 - *Pseudo* or *Virtual* Registers
 - unlimited number of registers
 - space is typically allocated on the stack with the stack frame
 - Hard Registers
 - Set of Registers Available in the Processor
 - Usually need to Obey some Usage Convention

Registers Name and Usage in RISC-V

#	Name	Usage
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporaries (Caller-save registers)
x6	t1	
x7	t2	
x8	s0/fp	Saved register / Frame pointer
x9	s1	Saved register
x10	a0	Function arguments / Return values
x11	a1	
x12	a2	Function arguments
x13	a3	
x14	a4	
x15	a5	

#	Name	Usage
x16	a6	Function arguments
x17	a7	
x18	s2	Saved registers (Callee-save registers)
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s8	
x25	s9	
x26	s10	
x27	s11	
x28	t3	Temporaries (Caller-save registers)
x29	t4	
x30	t5	
x31	t6	
	pc	Program counter

nbuter Architecture | Fall 2019 | lin-Soo Kim (linsoo.kim@snu.ac.kr)

Source: Jin-Soo Kim (linsoo.kim@snu.ac.kr), Systems Software & Architecture Lab., Seoul National University, Spring 2020

Basic RISC-V Instructions - Arithmetic

Instruction	Type	Example	Meaning
Add	R	add rd, rs1, rs2	$R[rd] = R[rs1] + R[rs2]$
Subtract	R	sub rd, rs1, rs2	$R[rd] = R[rs1] - R[rs2]$
Add immediate	I	addi rd, rs1, imm12	$R[rd] = R[rs1] + \text{SignExt}(\text{imm12})$
Set less than	R	slt rd, rs1, rs2	$R[rd] = (R[rs1] < R[rs2])? 1 : 0$
Set less than immediate	I	slti rd, rs1, imm12	$R[rd] = (R[rs1] < \text{SignExt}(\text{imm12}))? 1 : 0$
Set less than unsigned	R	sltu rd, rs1, rs2	$R[rd] = (R[rs1] <_u R[rs2])? 1 : 0$
Set less than immediate unsigned	I	sltiu rd, rs1, imm12	$R[rd] = (R[rs1] <_u \text{SignExt}(\text{imm12}))? 1 : 0$
Load upper immediate	U	lui rd, imm20	$R[rd] = \text{SignExt}(\text{imm20} \ll 12)$
Add upper immediate to PC	U	auipc rd, imm20	$R[rd] = \text{PC} + \text{SignExt}(\text{imm20} \ll 12)$

Basic RISC-V Instructions - Logic

Instruction	Type	Example	Meaning
AND	R	and rd, rs1, rs2	$R[rd] = R[rs1] \& R[rs2]$
OR	R	or rd, rs1, rs2	$R[rd] = R[rs1] \mid R[rs2]$
XOR	R	xor rd, rs1, rs2	$R[rd] = R[rs1] \wedge R[rs2]$
AND immediate	I	andi rd, rs1, imm12	$R[rd] = R[rs1] \& \text{SignExt}(imm12)$
OR immediate	I	ori rd, rs1, imm12	$R[rd] = R[rs1] \mid \text{SignExt}(imm12)$
XOR immediate	I	xori rd, rs1, imm12	$R[rd] = R[rs1] \wedge \text{SignExt}(imm12)$
Shift left logical	R	sll rd, rs1, rs2	$R[rd] = R[rs1] \ll R[rs2]$
Shift right logical	R	srl rd, rs1, rs2	$R[rd] = R[rs1] \gg R[rs2]$ (<i>logical</i>)
Shift right arithmetic	R	sra rd, rs1, rs2	$R[rd] = R[rs1] \gg R[rs2]$ (<i>arithmetic</i>)
Shift left logical immediate	I	slli rd, rs1, shamt	$R[rd] = R[rs1] \ll shamt$
Shift right logical imm.	I	srli rd, rs1, shamt	$R[rd] = R[rs1] \gg shamt$ (<i>logical</i>)
Shift right arithmetic immediate	I	srai rd, rs1, shamt	$R[rd] = R[rs1] \gg shamt$ (<i>arithmetic</i>)

Basic RISC-V Instr. – Data Transfer

Instruction	Type	Example	Meaning
Load doubleword	I	ld rd, imm12(rs1)	$R[rd] = \text{Mem}_8[R[rs1] + \text{SignExt}(imm12)]$
Load word	I	lw rd, imm12(rs1)	$R[rd] = \text{SignExt}(\text{Mem}_4[R[rs1] + \text{SignExt}(imm12)])$
Load halfword	I	lh rd, imm12(rs1)	$R[rd] = \text{SignExt}(\text{Mem}_2[R[rs1] + \text{SignExt}(imm12)])$
Load byte	I	lb rd, imm12(rs1)	$R[rd] = \text{SignExt}(\text{Mem}_1[R[rs1] + \text{SignExt}(imm12)])$
Load word unsigned	I	lwu rd, imm12(rs1)	$R[rd] = \text{ZeroExt}(\text{Mem}_4[R[rs1] + \text{SignExt}(imm12)])$
Load halfword unsigned	I	lhu rd, imm12(rs1)	$R[rd] = \text{ZeroExt}(\text{Mem}_2[R[rs1] + \text{SignExt}(imm12)])$
Load byte unsigned	I	lbu rd, imm12(rs1)	$R[rd] = \text{ZeroExt}(\text{Mem}_1[R[rs1] + \text{SignExt}(imm12)])$
Store doubleword	S	sd rs2, imm12(rs1)	$\text{Mem}_8[R[rs1] + \text{SignExt}(imm12)] = R[rs2]$
Store word	S	sw rs2, imm12(rs1)	$\text{Mem}_4[R[rs1] + \text{SignExt}(imm12)] = R[rs2](31:0)$
Store halfword	S	sh rs2, imm12(rs1)	$\text{Mem}_2[R[rs1] + \text{SignExt}(imm12)] = R[rs2](15:0)$
Store byte	S	sb rs2, imm12(rs1)	$\text{Mem}_1[R[rs1] + \text{SignExt}(imm12)] = R[rs2](7:0)$

Basic RISC-V Instr. – Control Transfer

Instruction	Type	Example	Meaning
Branch equal	SB	<code>beq rs1, rs2, imm12</code>	<code>if (R[rs1] == R[rs2]) pc = pc + SignExt(imm12 << 1)</code>
Branch not equal	SB	<code>bne rs1, rs2, imm12</code>	<code>if (R[rs1] != R[rs2]) pc = pc + SignExt(imm12 << 1)</code>
Branch greater than or equal	SB	<code>bge rs1, rs2, imm12</code>	<code>if (R[rs1] >= R[rs2]) pc = pc + SignExt(imm12 << 1)</code>
Branch greater than or equal unsigned	SB	<code>bgeu rs1, rs2, imm12</code>	<code>if (R[rs1] >= u R[rs2]) pc = pc + SignExt(imm12 << 1)</code>
Branch less than	SB	<code>blt rs1, rs2, imm12</code>	<code>if (R[rs1] < R[rs2]) pc = pc + SignExt(imm12 << 1)</code>
Branch less than unsigned	SB	<code>bltu rs1, rs2, imm12</code>	<code>if (R[rs1] < u R[rs2]) pc = pc + SignExt(imm12 << 1)</code>
Jump and link	UJ	<code>jal rd, imm20</code>	<code>R[rd] = PC + 4 PC = PC + SignExt(imm20 << 1)</code>
Jump and link register	I	<code>jalr rd, imm12(rs1)</code>	<code>R[rd] = PC + 4 PC = (R[rs1] + SignExt(imm12)) & (~1)</code>

Basic RISC-V Pseudo Instructions

Pseudo-instruction	Base instruction(s)	Meaning
li rd, imm	addi rd, x0, imm	Load immediate
la rd, symbol	auipc rd, D[31:12]+D[11] addi rd, rd, D[11:0]	Load absolute address where D = symbol - pc
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
bgt{u} rs, rt, offset	blt{u} rt, rs, offset	Branch if > (u: unsigned)
ble{u} rs, rt, offset	bge{u} rt, rs, offset	Branch if \geq (u: unsigned)
b{eq ne}z rs, offset	b{eq ne} rs, x0, offset	Branch if { = ≠ }
b{ge lt}z rs, offset	b{ge lt} rs, x0, offset	Branch if { \geq < }
b{le gt}z rs, offset	b{ge lt} x0, rs, offset	Branch if { \leq > }
j offset	jal x0, offset	Unconditional jump
call offset	jal ra, offset	Call subroutine (near)
ret	jalr x0, 0(ra)	Return from subroutine
nop	addi x0, x0, 0	No operation

Register Allocation Approaches

- Local Allocators: use Instruction-level knowledge
 - Top-Down: Use Frequency of Variables Use for Allocation
 - Bottom-Up: Evaluate Instructions Needs and Reuse Registers
- Global Allocators: use a Graph-Coloring Paradigm
 - Build a “conflict graph” or “interference graph”
 - Find a k -coloring for the graph, or change the code to a nearby problem that can be k -colored
- Common Algorithmic Trade-Off
 - Local Allocators are Fast
 - Some Problems with the Generated Code as they lack more Global Knowledge

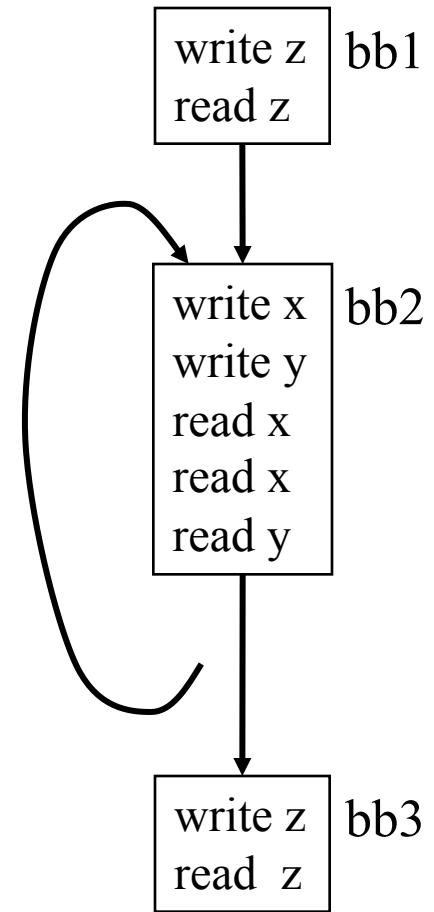
Local Register Allocation

- In General Hard Problem (*still*)
 - Code Generation for more than a single Register is NP-Complete
- Use Simple Strategies:
 - Top-Down: Just put in Register Names that Occur more Often
 - Bottom-Up: Evaluate each Instruction and Keep Track of When Values are Needed Later On.
- Extension to Multiple Basic Blocks
 - Using Profile Data to Determine Frequently Executed Paths
 - Use Nesting Depth of Code

Top-Down Local Register Allocator

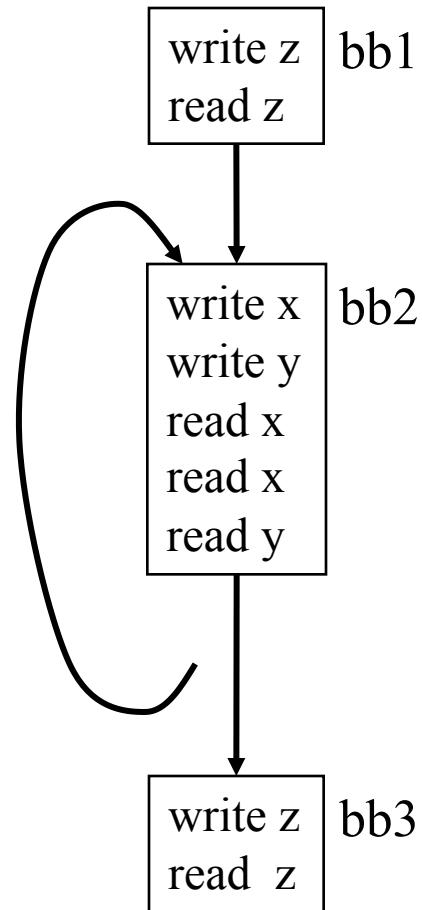
- Estimate the Benefits of Putting each Variable in a Register in a Particular Basic Block
 - $\text{Benefit}(V, B) = \text{Number of } \textit{uses} \text{ and } \textit{defs} \text{ of the var } V \text{ in basic block } B$
- Estimate the Overall Benefit
 - $\text{TotBenefit}(V) = \text{Benefit}(V, B) * \text{freq}(B)$ for all basic block B
 - If $\text{freq}(B)$ is not known, use 10^{depth} where depth represents the nesting depth of B in the CFG of the code.
- Assign the (R-*feasible*) Highest-payoff Variables to Registers
 - Reserve *feasible* registers for basic calculations and evaluation.
 - Rewrite the code inserting load/store operation where appropriate.

Example



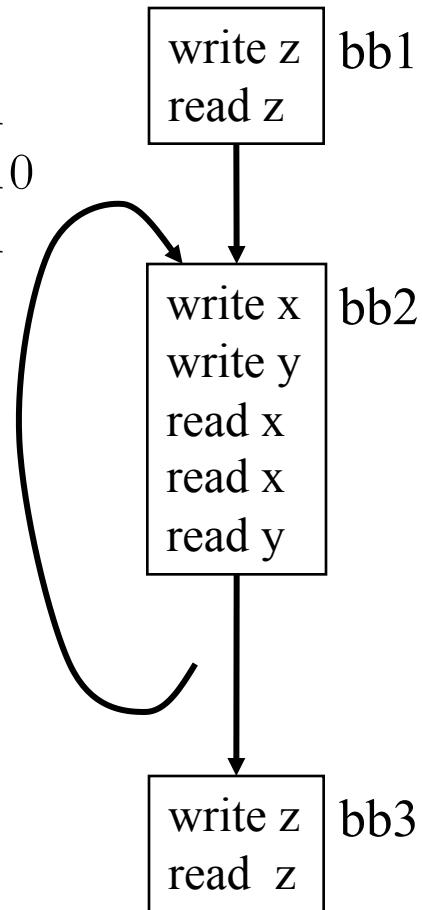
Example

- Benefit for basic blocks
 - $\text{Benefit}(x, \text{bb1}) = 0$
 - $\text{Benefit}(x, \text{bb2}) = 3$
 - $\text{Benefit}(x, \text{bb3}) = 0$
 - $\text{Benefit}(y, \text{bb1}) = 0$
 - $\text{Benefit}(y, \text{bb2}) = 2$
 - $\text{Benefit}(y, \text{bb3}) = 0$
 - $\text{Benefit}(z, \text{bb1}) = 2$
 - $\text{Benefit}(z, \text{bb2}) = 0$
 - $\text{Benefit}(z, \text{bb3}) = 2$



Example

- Benefit for basic blocks
 - $\text{Benefit}(x, \text{bb1}) = 0$
 - $\text{Benefit}(x, \text{bb2}) = 3$
 - $\text{Benefit}(x, \text{bb3}) = 0$
 - $\text{Benefit}(y, \text{bb1}) = 0$
 - $\text{Benefit}(y, \text{bb2}) = 2$
 - $\text{Benefit}(y, \text{bb3}) = 0$
 - $\text{Benefit}(z, \text{bb1}) = 2$
 - $\text{Benefit}(z, \text{bb2}) = 0$
 - $\text{Benefit}(z, \text{bb3}) = 2$
- Frequency
 - $\text{freq}(\text{bb1}) = 1$
 - $\text{freq}(\text{bb2}) = 10$
 - $\text{freq}(\text{bb3}) = 1$

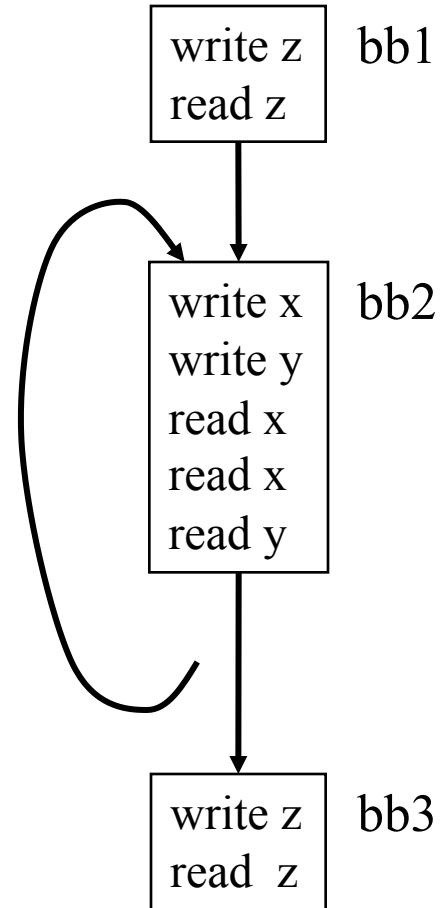


Example

- Benefit for basic blocks
 - $\text{Benefit}(x, \text{bb1}) = 0$
 - $\text{Benefit}(x, \text{bb2}) = 3$
 - $\text{Benefit}(x, \text{bb3}) = 0$
 - $\text{Benefit}(y, \text{bb1}) = 0$
 - $\text{Benefit}(y, \text{bb2}) = 2$
 - $\text{Benefit}(y, \text{bb3}) = 0$
 - $\text{Benefit}(z, \text{bb1}) = 2$
 - $\text{Benefit}(z, \text{bb2}) = 0$
 - $\text{Benefit}(z, \text{bb3}) = 2$
 - Frequency
 - $\text{freq(bb1)} = 1$
 - $\text{freq(bb2)} = 10$
 - $\text{freq(bb3)} = 1$
 - Total Benefit
 - $\text{TotBenefit}(x) = 0*1 + 3*10 + 0*1 = 30$
 - $\text{TotBenefit}(y) = 0*1 + 2*10 + 0*1 = 20$
 - $\text{TotBenefit}(z) = 2*1 + 0*10 + 2*1 = 4$
-
- ```
graph TD; bb1[write z
read z] --> bb2[write x
write y
read x
read x
read y]; bb2 --> bb3[write z
read z]
```

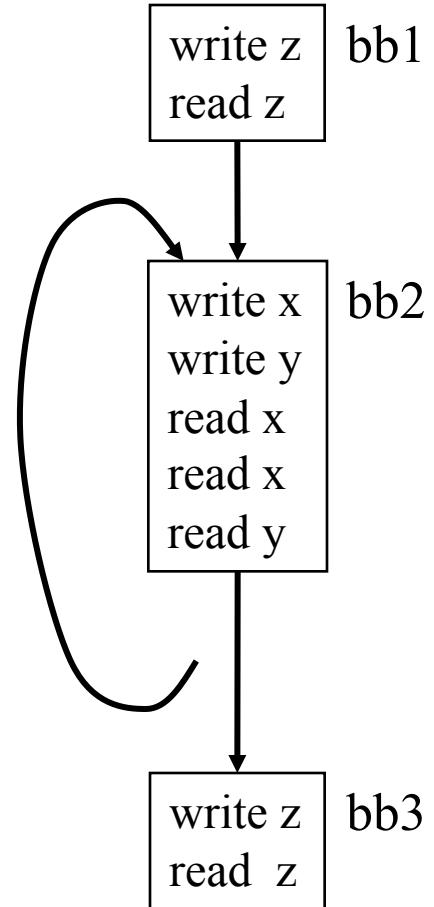
# Example

- Total Benefit
  - $\text{TotBenefit}(x) = 30$
  - $\text{TotBenefit}(y) = 20$
  - $\text{TotBenefit}(z) = 4$
- Assume 2 Registers are Available
  - Assign x and y to Registers



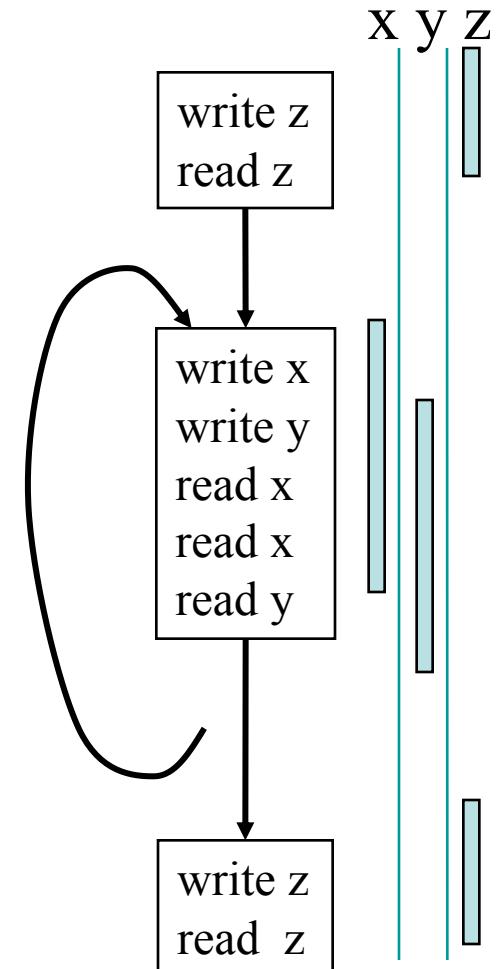
# Problem 1

- Allocation is same as above
  - x, and y get registers, but not z
- The variables need to occupy the registers even when it does not need it
- All x, y and z can have registers



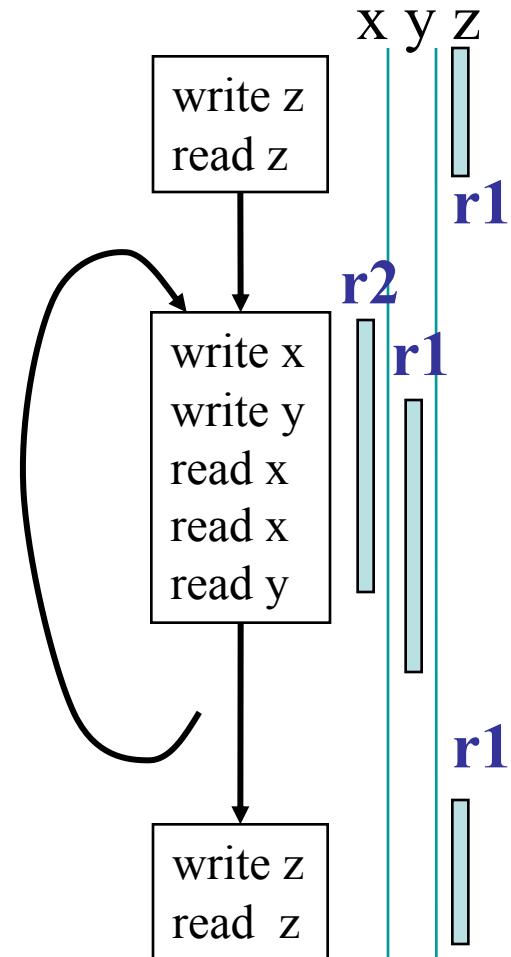
# Problem 1

- Allocation is same as above
  - x, and y get registers, but not z
- The variables need to occupy the registers even when it does not need it
- All x, y and z can have registers



# Problem 1

- Allocation is same as above
  - x, and y get registers, but not z
- The variables need to occupy the registers even when it does not need it
- All x, y and z can have registers



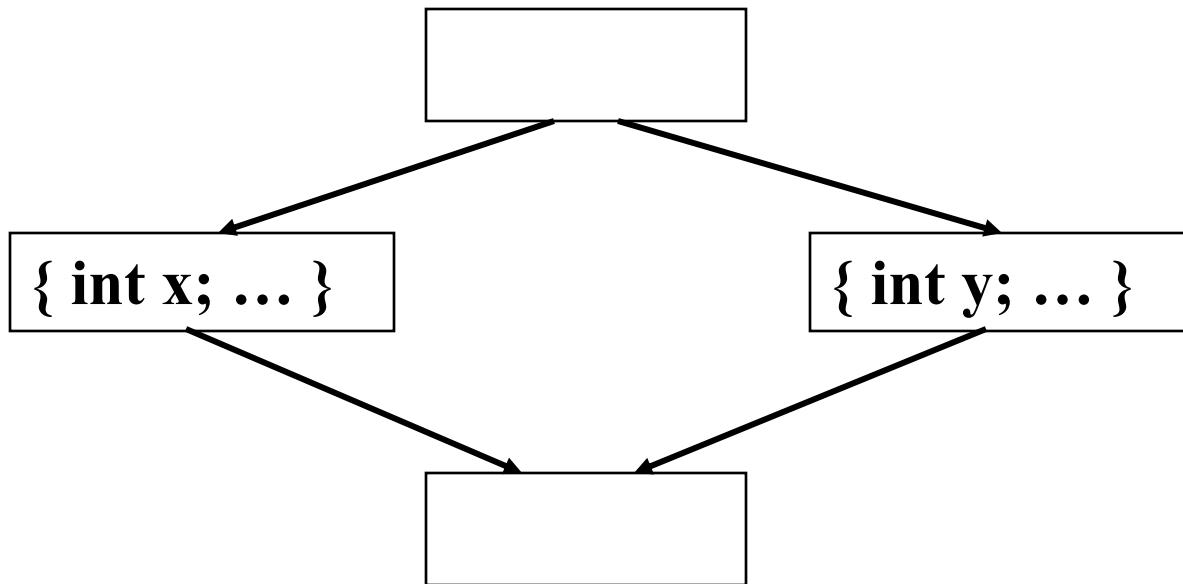
# Problem 2

---

- Even non-interfering variables don't share registers

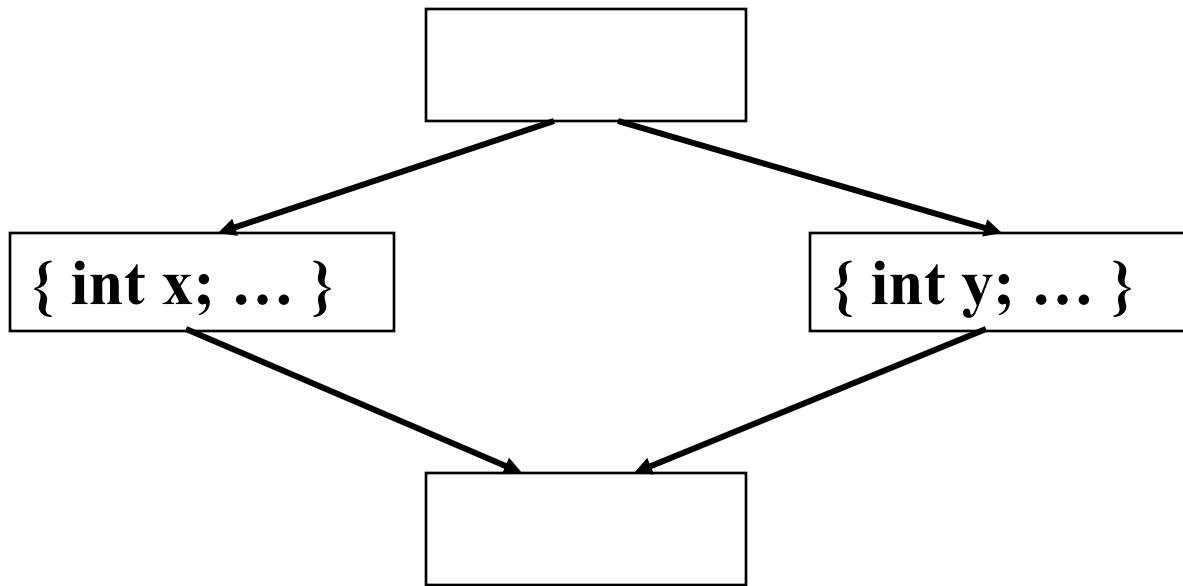
# Problem 2

- Even non-interfering variables don't share registers



# Problem 2

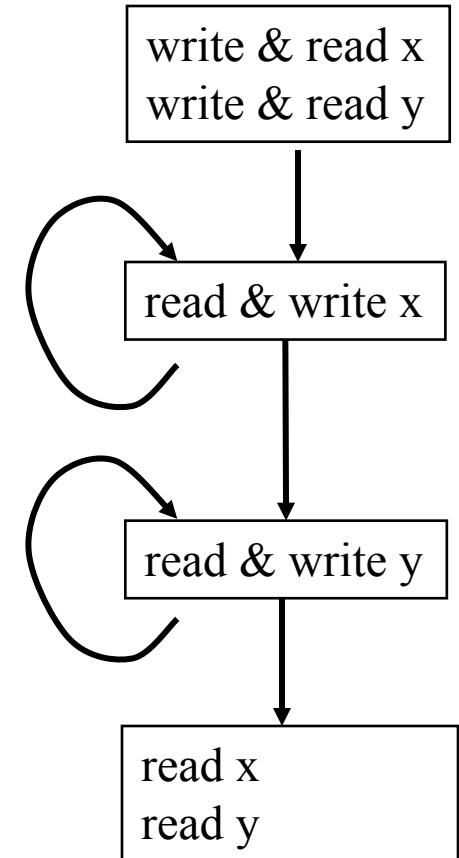
- Even non-interfering variables don't share registers



- x and y can use the same registers

# Problem 3

- Different phases of the program behave differently
- One register available
  - Register for x in the first loop
  - Register for y in the second loop
  - Don't care too much about the rest
- Need to spill
  - Top-Down “All or Nothing” will not work



# Bottom-Up Local Allocator

---

- Basic Idea:
  - Focus on the Needs of Each Instructions in a Basic Block
    - Ensure Each Instruction Can Execute
    - Instruction Operands and Results in Registers
  - Transitions Between Instructions
    - Observe Which Values are Used Next - in the Future
- On-Demand Allocation:
  - Iterate Through the Instructions of a **Basic Block**
  - Allocate the Value of the Operand, if Not Already in a Register
  - Allocate Register for Result
  - When Out of Registers:
    - Release Register Whose Value is to be Used *Farthest* into the Future
    - Dirty Register Value Requires Memory Operation to Update Storage

# Basic Block

---

- Def: A *Maximal Sequence* of Instructions such that:
  1. Only the first instruction can be reached from outside the basic block
  2. All the instructions are executed consecutively *iff* the first instruction is executed
    - No branch or jump instructions in the basic block
    - Except the last instruction
    - No labels within the basic block
    - Except before the first instruction

# Basic Block - Algorithm

---

- **Input:** Sequence of Three-Address Instructions
- **Output:** A list of Basic Blocks
- **Algorithm:**
  - Determine the set of *leader instructions* - the head of each basic block - using the following:
    - The first statement of the program is a *leader*
    - Any statement that is the target of a goto (either conditional or not) is a *leader instructions*
    - Any statement that immediately follows a goto or unconditional goto statement is a *leader instruction*
  - For each *leader instruction*, its basic block consists of the *leader instruction* and all the statements up to but not including the next *leader instruction* or the end of the program.

# Basic Block - Example

```
Sum: a = arg0
 b = arg1
 c = 0
 if (a <= 0) goto L2
 i = arg0
L1: i = i - 1
 a = b[i]
 b = b + 4
 c = c + a
 if (i >= 0) goto L1
L2: ret c
```

## Step 1: Detect Leader Instructions

- The first statement of the program is a *leader*
- Any statement that is the target of a goto (either conditional or not) is a *leader instructions*
- Any statement that immediately follows a goto or unconditional goto statement is a *leader instruction*

## Step 2: Determine Block Instructions

- For each *leader instruction*, its basic block consists of the *leader instruction* and all the statements up to but not including the next *leader instruction* or the end of the program

# Basic Block - Example

```
Sum: a = arg0
 b = arg1
 c = 0
 if (a <= 0) goto L2
 i = arg0
L1: i = i - 1
 a = b[i]
 b = b + 4
 c = c + a
 if (i >= 0) goto L1
L2: ret c
```

## Step 1: Detect Leader Instructions

- The first statement of the program is a *leader*
- Any statement that is the target of a goto (either conditional or not) is a *leader instructions*
- Any statement that immediately follows a goto or unconditional goto statement is a *leader instruction*

## Step 2: Determine Block Instructions

- For each *leader instruction*, its basic block consists of the *leader instruction* and all the statements up to but not including the next *leader instruction* or the end of the program

# Basic Block - Example

```
Sum: a = arg0
 b = arg1
 c = 0
 if (a <= 0) goto L2
 i = arg0

L1: i = i - 1
 a = b[i]
 b = b + 4
 c = c + a
 if (i >= 0) goto L1
 ret c

L2:
```

## Step 1: Detect Leader Instructions

- The first statement of the program is a *leader*
- Any statement that is the target of a goto (either conditional or not) is a *leader instructions*
- Any statement that immediately follows a goto or unconditional goto statement is a *leader instruction*

## Step 2: Determine Block Instructions

- For each *leader instruction*, its basic block consists of the *leader instruction* and all the statements up to but not including the next *leader instruction* or the end of the program

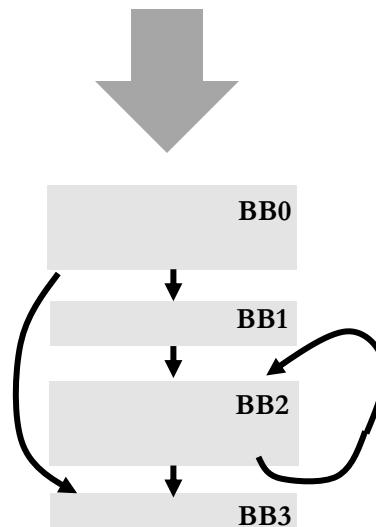
# Basic Block - Example

```

Sum: a = arg0
 b = arg1
 c = 0
 if (a <= 0) goto L2
 i = arg0

L1: i = i - 1
 a = b[i]
 b = b + 4
 c = c + a
 if (i >= 0) goto L1
 ret c

```



## Step 1: Detect Leader Instructions

- The first statement of the program is a *leader*
- Any statement that is the target of a goto (either conditional or not) is a *leader instructions*
- Any statement that immediately follows a goto or unconditional goto statement is a *leader instruction*

## Step 2: Determine Block Instructions

- For each *leader instruction*, its basic block consists of the *leader instruction* and all the statements up to but not including the next *leader instruction* or the end of the program

## Graph of Basic Blocks

- Define the Control-Flow of a Program
- *More Later...*

# Bottom-Up Local Allocator

- Details:
  - Instructions in format:  $vr_z \leftarrow vr_x \text{ op } vr_y$  using virtual registers or temps
  - Data Structures:
    - The Number of Physical Registers Available
    - The Virtual Name associated with Each Physical Register
    - For Each Virtual Name a Distance to the Next Use in the Basic Block
    - A Flag Indicating if the Corresponding Physical Register is in Free or in Use
    - A Stack of Free Physical Registers with a Stack Pointer (Integer Index)
  - Auxiliary Functions:
    - `ensure(vrx)` allocates a physical register to ensure storage for  $vr_x$
    - `free(vrx)` releases the physical register holding  $vr_x$
    - `allocate(vrx)` just sets some flags claiming register being used.
    - `dist(vrx)` returns the distance to the next reference to  $vr_x$  in the basic block.

```
initialize(size)
Size ← size;
for i ← size-1 to 0 do
 Name[i] ← -1;
 Next[i] ← ∞;
 Free[i] ← true;
 push(i);
StackTop = size-1;
```

# Bottom-Up Local Allocator

## Bottom-Up Local Allocation Algorithm

*Input:* Basic Block B

*Output:* Rewritten Instruction in B

```

foreach instr i: vri3 ← vri1 op vri2 ∈ B do

 rx ← ensure(vri1);
 ry ← ensure(vri2);
 if(vri1 is not needed after i)then
 free(vri1);
 if(vri2 is not needed after i)then
 free(vri2);
 rz ← allocate(vri3);

 rewrite i as rz ← rx op ry

 if (vri1 is needed after i) then
 Next[rx] = dist(vri1);
 else
 Next[rx] = ∞;
 if (vri2 is needed after i) then
 Next[ry] = dist(vri2);
 else
 Next[ry] = ∞;
 Next[rz] = dist(vri3);
end

```

```

void free(i)
 if(Free[i]≠ true) then
 push(i);
 Name[i] ← -1;
 Next[i] ← ∞;
 Free[i] ← true;
 end

reg ensure(vr)
 r ← find(vr);
 if(r exists) then
 result ← r;
 else
 result ← allocate(vr);
 emit code to move vr into r;
 end
 return result;

reg allocate(vr)
 if(StackTop ≥ 0) then
 r ← pop();
 else
 r ← findMaxNext();
 if(r is dirty) then
 save r in memory;
 Name[r] ← vr;
 Next[r] ← -1;
 Free[r] ← false;
 return r;

```

# Bottom-Up Local Allocator

## Bottom-Up Local Allocation Algorithm

*Input:* Basic Block B

*Output:* Rewritten Instruction in B

```

foreach instr i: $vr_{i3} \leftarrow vr_{i1} \text{ op } vr_{i2} \in B$ do

 $r_x \leftarrow \text{ensure}(vr_{i1});$
 $r_y \leftarrow \text{ensure}(vr_{i2});$
 if(vr_{i1} is not needed after i) then
 free(vr_{i1});
 if(vr_{i2} is not needed after i) then
 free(vr_{i2});
 $r_z \leftarrow \text{allocate}(vr_{i3});$

 rewrite i as $r_z \leftarrow r_x \text{ op } r_y$

 if (vr_{i1} is needed after i) then
 Next[r_x] = dist(vr_{i1});
 else
 Next[r_x] = ∞ ;
 if (vr_{i2} is needed after i) then
 Next[r_y] = dist(vr_{i2});
 else
 Next[r_y] = ∞ ;
 Next[r_z] = dist(vr_{i3});
end

```

Next field is temporarily set to -1 so that a possible second allocate will not bump this register on the same instruction

```

void free(i)
 if(Free[i] ≠ true) then
 push(i);
 Name[i] ← -1;
 Next[i] ← ∞ ;
 Free[i] ← true;
 end

reg ensure(vr)
 r ← find(vr);
 if(r exists) then
 result ← r;
 else
 result ← allocate(vr);
 emit code to move vr into r;
 end
 return result;

reg allocate(vr)
 if(StackTop ≥ 0) then
 r ← pop();
 else
 r ← findMaxNext();
 if(r is dirty) then
 save r in memory;
 Name[r] ← vr;
 Next[r] ← -1;
 Free[r] ← false;
 end
 return r;

```

# Bottom-Up Allocator Example

---

```
vr3 ← vr1 op vr2
vr5 ← vr4 op vr1
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```

# Bottom-Up Allocator Example

```
vr3 ← vr1 op vr2
vr5 ← vr4 op vr1
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```

size = 3

|       | 0  | 1  | 2       |
|-------|----|----|---------|
| Name  | -1 | -1 | -1      |
| Next  | ∞  | ∞  | ∞       |
| Free  | T  | T  | T       |
| Stack | 2  | 1  | 0       |
|       |    |    | Top = 2 |

# Bottom-Up Allocator Example

```
mem → r0 // vr1's value
vr3 ← r0 op vr2
vr5 ← vr4 op vr1
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```

Size = 3

|       | 0               | 1  | 2       |
|-------|-----------------|----|---------|
| Name  | vr <sub>1</sub> | -1 | -1      |
| Next  | 1               | ∞  | ∞       |
| Free  | F               | T  | T       |
| Stack | 2               |    |         |
|       |                 |    | Top = 1 |
|       | 1               |    |         |

# Bottom-Up Allocator Example

```
mem → r0 // vr1's value
mem → r1 // vr2's value
vr3 ← r0 op r1
vr5 ← vr4 op vr1
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```

Size = 3

|       | 0               | 1               | 2       |
|-------|-----------------|-----------------|---------|
| Name  | vr <sub>1</sub> | vr <sub>2</sub> | -1      |
| Next  | 1               | 3               | ∞       |
| Free  | F               | F               | T       |
| Stack | 2               |                 |         |
|       |                 |                 | Top = 0 |

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
vr5 ← vr4 op vr1
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```

Size = 3

|       | 0               | 1               | 2               |
|-------|-----------------|-----------------|-----------------|
| Name  | vr <sub>1</sub> | vr <sub>2</sub> | vr <sub>3</sub> |
| Next  | 1               | 3               | 3               |
| Free  | F               | F               | F               |
| Stack |                 |                 |                 |
|       |                 |                 | Top = -1        |

# Bottom-Up Allocator Example

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
vr5 ← vr4 op vr1
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```

Run Out of Registers for vr<sub>4</sub>

Allocate will find register with maximum value of Next to Spill to Memory, e.g., vr<sub>2</sub>

Size = 3

|       | 0               | 1               | 2               |
|-------|-----------------|-----------------|-----------------|
| Name  | vr <sub>1</sub> | vr <sub>2</sub> | vr <sub>3</sub> |
| Next  | 1               | 3               | 3               |
| Free  | F               | F               | F               |
| Stack |                 |                 |                 |

Top = -1

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
vr5 ← r1 op vr1
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```

Size = 3

|       |                 |                 |                 |
|-------|-----------------|-----------------|-----------------|
|       | 0               | 1               | 2               |
| Name  | vr <sub>1</sub> | vr <sub>4</sub> | vr <sub>3</sub> |
| Next  | 1               | -1              | 3               |
| Free  | F               | F               | F               |
| Stack |                 |                 |                 |
|       |                 |                 | Top = -1        |

# Bottom-Up Allocator Example

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
vr5 ← r1 op vr1
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```

Size = 3

|       | 0               | 1        | 2               |
|-------|-----------------|----------|-----------------|
| Name  | vr <sub>1</sub> | -1       | vr <sub>3</sub> |
| Next  | 1               | $\infty$ | 3               |
| Free  | F               | T        | F               |
| Stack | 1               |          |                 |

Top = 0

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
vr5 ← r1 op r0
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```

Size = 3

|       |    |    |                 |
|-------|----|----|-----------------|
|       | 0  | 1  | 2               |
| Name  | -1 | -1 | vr <sub>3</sub> |
| Next  | ∞  | ∞  | 3               |
| Free  | T  | T  | F               |
| Stack | 1  |    |                 |
|       | 0  |    |                 |

Top = 1

# Bottom-Up Allocator Example

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
r0 ← r1 op r0
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```

Size = 3

|       | 0               | 1        | 2               |
|-------|-----------------|----------|-----------------|
| Name  | vr <sub>5</sub> | -1       | vr <sub>3</sub> |
| Next  | 1               | $\infty$ | 3               |
| Free  | F               | T        | F               |
| Stack | 1               |          |                 |

Top = 0

# Bottom-Up Allocator Example

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
r0 ← r1 op r0
vr6 ← r0 op vr6
vr7 ← vr3 op vr2
```

Size = 3

|       | 0               | 1  | 2               |
|-------|-----------------|----|-----------------|
| Name  | vr <sub>5</sub> | -1 | vr <sub>3</sub> |
| Next  | -1              | ∞  | 3               |
| Free  | F               | T  | F               |
| Stack | 1               |    |                 |
|       |                 |    | Top = 0         |

# Bottom-Up Allocator Example

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
r0 ← r1 op r0
mem → r1 // vr6's value
vr6 ← r0 op r1
vr7 ← vr3 op vr2
```

Size = 3

|       | 0               | 1               | 2               |
|-------|-----------------|-----------------|-----------------|
| Name  | vr <sub>5</sub> | vr <sub>6</sub> | vr <sub>3</sub> |
| Next  | -1              | -1              | 3               |
| Free  | F               | F               | F               |
| Stack |                 |                 |                 |
|       |                 |                 | Top = -1        |

# Bottom-Up Allocator Example

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
r0 ← r1 op r0
mem → r1 // vr6's value
r1 ← r0 op r1
vr7 ← vr3 op vr2
```

Size = 3

|       | 0               | 1               | 2               |
|-------|-----------------|-----------------|-----------------|
| Name  | vr <sub>5</sub> | vr <sub>6</sub> | vr <sub>3</sub> |
| Next  | -1              | -1              | 3               |
| Free  | F               | F               | F               |
| Stack |                 |                 |                 |
|       |                 |                 | Top = -1        |

# Bottom-Up Allocator Example

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
r0 ← r1 op r0
mem → r1 // vr6's value
r1 ← r0 op r1
vr7 ← vr3 op vr2
```

Size = 3

|       | 0  | 1  | 2               |
|-------|----|----|-----------------|
| Name  | -1 | -1 | vr <sub>3</sub> |
| Next  | ∞  | ∞  | 3               |
| Free  | T  | T  | F               |
| Stack | 0  |    |                 |
|       | 1  |    |                 |

Top = 1

# Bottom-Up Allocator Example

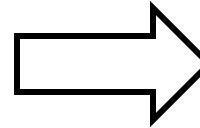
```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
r0 ← r1 op r0
mem → r1 // vr6's value
r1 ← r0 op r1
r1 → mem // vr6's value
r0 ← r2 op r1
```

Size = 3

|       | 0               | 1               | 2               |
|-------|-----------------|-----------------|-----------------|
| Name  | vr <sub>7</sub> | vr <sub>6</sub> | vr <sub>3</sub> |
| Next  | -1              | -1              | -1              |
| Free  | F               | F               | F               |
| Stack |                 |                 |                 |
|       |                 |                 | Top = -1        |

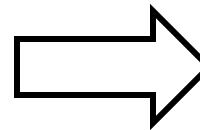
# Bottom-Up Allocator Example

```
vr3 ← vr1 op vr2
vr5 ← vr4 op vr1
vr6 ← vr5 op vr6
vr7 ← vr3 op vr2
```



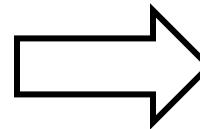
```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
r0 ← r1 op r0
mem → r1 // vr6's value
r1 ← r0 op r1
r1 → mem // vr6's value
mem → r1 // vr2's value
r0 ← r2 op r1
```

```
t3 ← t1 op t2
t5 ← t4 op t1
t6 ← t5 op t6
t7 ← t3 op t2
```



```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
r0 ← r1 op r0
mem → r1 // vr6's value
r1 ← r0 op r1
r1 → mem // vr6's value
mem → r1 // vr2's value
r0 ← r2 op r1
```

```
t3 ← a + b
t5 ← c * a
d ← t5 - d
t7 ← t3 + b
```



```
lw t1, 8(sp) // a on the Stack
lw t2, 16(sp) // b on the Stack
add t3, t1, t2 // t3 is add
sw t2, 16(sp) // b to stack
lw t2, 24(sp) // c on the Stack
mul t1, t2, t1 // t1 is mult
lw t2, 32(sp) // d on the stack
sub t2, t1, t2 // t2 is sub
sw t2, 32(sp) // d to stack
lw t2, 16(sp) // b on the stack
add t1, t3, t2 // t1 is add
```

# Dirty and Clean Registers

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
r0 ← r1 op r0
mem → r1 // vr6's value
r1 ← r0 op r1
r1 → mem // vr6's value
mem → r1 // vr2's value
r0 ← r2 op r1
```

```
lw t1, 8(sp) // a on the Stack
lw t2, 16(sp) // b on the Stack
add t3, t1, t2 // t3 is add
sw t2, 16(sp) // b to stack
lw t2, 24(sp) // c on the Stack
mul t1, t2, t1 // t1 is mult
lw t2, 32(sp) // d on the stack
sub t2, t1, t2 // t2 is sub
sw t2, 32(sp) // d to stack
lw t2, 16(sp) // b on the stack
add t1, t3, t2 // t1 is add
```

- Choosing which Register to Reuse:
  - Dirty - Need to Update Memory
  - Clean - Just Reuse the Physical Register
- Idea: Give Preference to Clean Registers
  - No Need to Save Contents to Memory
- Not Always the Best Approach:
  - When Dirty Value is Reused Far Way
  - Better to Restore it to Memory
  - Rather than Holding on to the Register

# Dirty and Clean Registers

```
mem → r0 // vr1's value
mem → r1 // vr2's value
r2 ← r0 op r1
r1 → mem // vr2's value
mem → r1 // vr4's value
r0 ← r1 op r0
mem → r1 // vr6's value
r1 ← r0 op r1
r1 → mem // vr6's value
mem → r1 // vr2's value
r0 ← r2 op r1
```

```
lw t1, 8(sp) // a on the Stack
lw t2, 16(sp) // b on the Stack
add t3, t1, t2 // t3 is add
sw t2, 16(sp) // b to stack
lw t2, 24(sp) // c on the Stack
mul t1, t2, t1 // t1 is mult
lw t2, 32(sp) // d on the stack
sub t2, t1, t2 // t2 is sub
sw t2, 32(sp) // d to stack
lw t2, 16(sp) // b on the stack
add t1, t3, t2 // t1 is add
```

- Choosing which Register to Reuse:
  - Dirty - Need to Update Memory
  - Clean - Just Reuse the Physical Register
- Idea: Give Preference to Clean Registers
  - No Need to Save Contents to Memory
- Not Always the Best Approach:
  - When Dirty Value is Reused Far Way
  - Better to Restore it to Memory
  - Rather than Holding on to the Register

# What a Smart Allocator Needs to Do

---

- Determine ranges for each variable can benefit from using a register (Webs)
- Determine which of these ranges overlap (Interference)
- Find the benefit of keeping each web in a register (Spill Cost)
- Decide which webs gets a register (Allocation)
- Split Webs if needed (Spilling and Splitting)
- Assign hard registers to webs (Assignment)
- Generate code including spills (Code Generation)

# Summary

---

- Register Allocation and Assignment
  - Very Important Transformations and Optimization
  - In General Hard Problem (NP-Complete)
  
- Many Approaches
  - Local Methods: Top-Down and Bottom-Up
  - Quick but not Necessarily Very Good