Final Code Generation

- Input: Intermediate representation of program, Output: Assembly instructions
- General approach:
  - Interpret IR: examples Java byte code. Usually slower.
  - Macro expansion: replace each IR by a set of target machine instructions.
    - Naive and can be inefficient.
    - Does not utilize the state of program to generate efficient program.
    - Sometimes more than one IR can be replaced by a single instruction.
    - Possible code bloating.
  - Threaded code:
    - Replace each IR instruction with a subroutine call to a support routine that implements the IR instruction.
    - After executing the instruction, an implementation routine uses the return address to select the next IR.
    - Control threaded through a sequence of calls to implementation routines.
- We will look at code generation issues:
  1. Instruction selection
  2. Register and Temporary Management
  3. Instruction scheduling

Instruction selection

- Uniformity and completeness of instruction set important during code generation.
- Efficiency an important consideration (not in the project, hence mapping is straightforward).
- Simple Approach: For each three-address statement, find a code skeleton that implements it efficiently:

```
X:=Y + Z            lw $19, Y
                    lw $20, Z
                    add $21, $19, $20
                    sw $21, X
```

- Generates poor code:

```
a:=b + c;           lw $19, b
                    lw $20, c
                    add $21, $19, $20
                    sw $21, a
d:=a + e;           lw $19, a
                    lw $20, e
                    add $21, $19, $20
                    sw $21, d
```

  Loads and stores of a not required.
- Efficiency an important issues: if target machine provides many instructions for implementing an operations, cost issues come into picture.
  For instance, instruction `a:=a + 1` can be implemented by INC instruction: generally very efficient.
  Deciding which sequence is best may require knowledge about the context in which a construct appears.

Register Allocation

- Temporaries and registers used to hold intermediate results.
- Instructions involving registers are faster (no need to access memory). In most RISC processors, computations occur only through registers.
- Optimization problem: efficiently allocate registers to minimize execution time. *Constraints:*
  - Fixed number of registers $\Rightarrow$ make good use of them
  - H/W and OS may require that register usage convention be observed.
- Two subproblems:
  1. Register allocation: pick variables for reg. allocation
  2. Register assignment: map variables to specific registers.
- Finding optimal assignment of registers to variables is difficult (in general, NP-complete).
- Register pool divided into three kinds:
  1. Allocatable registers: Explicitly allocated and freed by compile-time calls to register management routines.
  2. Res. registers: Assigned a fixed function throughout a program. Examples: display, stack pointer, frame-pointer etc.
  3. Volatile registers: Can be safely used only in local code sequence. Used mostly for i) holding values of variables when implementing A:= B kind of statement, ii) holding index or offset into some table etc.

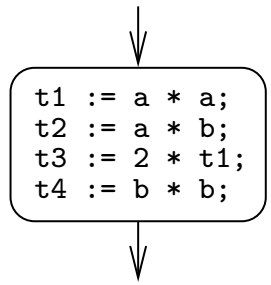
Allocation of registers

- Simplest possible scheme: load-use-store model. Every time a variable is accessed, it is loaded into registers, used, and then value stored back.

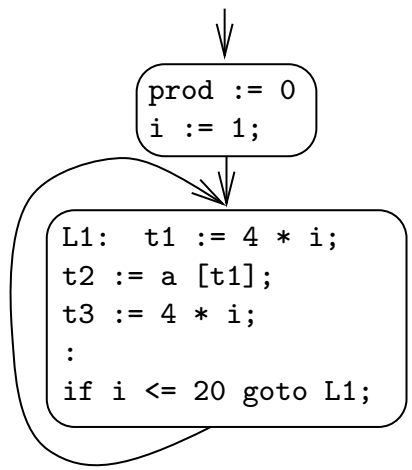
  Used in your project, but can be very inefficient.
- Smarter scheme: Keep results and data value in registers if results and data value are going to be used again. This means that code generator must keep track of following:
  - Which registers are in use and what they hold?
  - Where the current value of a variable is to be found?
  - Which variable will be needed later and where?
  - Which variables whose current values are in registers need to be stored in memory beforing calling procedures, making jumps etc.
- Approach:
  - Limit above analysis to variables and temps within *blocks*
  - Analyze and record *liveness* of variables and temps
  - Extend symbol table to store information about variables where they are in memory
  - Store information about registers as well.
  - Use above information for register allocation and code generation.
- We will look at all of these in next few slides.

Step 1: Determine blocks in IR

- Flow graph: A graph based representation of IR programs. Captures information as to how information/control flows.
- Node: computations, Edges: control flow
- Basic block: sequence of statements in which flow of control enter at beginning and leaves at end.
  - *Define* a variable x: value written into x
  - *Use* of x: value of x read.
  - *live*: value of a variable is live at a *point* if it is used after that point.



- Given a sequence of statements, convert into a graph with basic blocks connected through control flows:
  1. Find set of all first statements, called leader, of a block:
     - first statement of program is a leader
     - Target of a gotos is a leader
     - Statement following a goto is a leader.
  2. A basic block = leader + all statements until next leader.



Step 1: transformation of basic blocks

- Transform statements within a block so that better code will be produced. Two kinds: i) Structure preserving ii) Algebraic
- Structure preserving transformations:
  1. Common subexpression elimination: Compute common expressions only once.

Original	Transformed
`a:=b + c`	`a:=b + c`
`b:=a - d`	`b:=a - d`
`c:=b + c`	`c:=b + c`
`d:=a -d`	`d:=b`

  2. Dead-code elimination: if value of `x` is never used, then one can eliminate statements of the form `x:=y + z;`
  3. Rename temporary variables.
  4. Interchange of statements:

Original	Transformed
`t1:=b + c`	`t2:=a - d`
`t2:=a - d`	`t1:=b + c`

- Algebraic: Perform simple algebraic transformations such as multiply by 0 or 1 etc.

Step 2: find liveness and usage of variables

- Use of a variable `x`:

```
x:=a + b;          (statement i)
   :
y:=x + c;          (statement j)
```

  `x` is used in statement `j`.
- Next use of a variable:

```
a:=b + c;          (statement i)
   :
k:=b + d;          (statement j)
```

  If b is not used between statement i and j, next-use of b at statement i is statement j.
- A variable is *live* if it is going to be used again in a program.
  In the absence of information, assume that programmer-defined variables are live at the end of block and temporaries are not.
- Question: how to compute next-use of variables. Start from end of block and work backwards to construct it.

Step 2: find next-usage of variables

- Extend symbol table to hold two extra information: i) *status* that indicates if variable is alive or dead, ii) *next-use* that indicates statement where variable is used.
- Make *status* of all variables *live* and all temporaries *dead*.
- Make *next-use* of all variables and temporaries none.
- Start at the end of a basic block: better to scan from the back.
- For instruction `(j) a:=b + c`
  1. Check symbol table and see whether `a`, `b`, or `c` are alive or dead, and see what their next use is. Attach this information to this instruction.
  2. Update symbol-table information for use earlier in block:
     - 2.1 Make the symbol table entry for a dead and no next-use
     - 2.2 mark symbol table entries for `b` and `c` live and set their next-use values to line j.
- Example: For block:

```
(1) u:=a - b
(2) v:=c - a
(3) w:=u + v
(4) x:=d + b
(5) y:=c + 1
(6) z:=x * y
(7) d:=w - z
```

- Next-use information can be used for managing temporaries: two temporaries can be stored in same location if they are not live at the same time.

Example: computation of next-use

Var	Status	next-use
a	Live	None
b	Live	None
c	Live	None
d	Live	None
u	Dead	None
v	Dead	None
w	Dead	None
x	Dead	None
y	Dead	None
z	Dead	None

No	Instr	Status	next-use
1	u:=a - b		
2	v:=c - a		
3	w:=u + v		
4	x:=d + b		
5	y:=c + 1		
6	z:=x * y		
7	d:=w - z	d live; w,z:dead	d, w,z: none

Var	Status	next-use
a	Live	None
b	Live	None
c	Live	None
d	**Dead**	**None**
u	Dead	None
v	Dead	None
w	**Live**	**(7)**
x	Dead	None
y	Dead	None
z	**Live**	**(7)**

No	Instr	Status	next-use
1	u:=a - b		
2	v:=c - a		
3	w:=u + v		
4	x:=d + b		
5	y:=c + 1		
6	z:=x * y	z live; x,y:dead	z:(7), x,y:none
7	d:=w - z	d live; w,z:dead	d, w,z: none

Var	Status	next-use
a	Live	None
b	Live	None
c	Live	None
d	Dead	None
u	Dead	None
v	Dead	None
w	Live	(7)
x	**Live**	**(6)**
y	**live**	**(6)**
z	**Dead**	**None**

No	Instr	Status	next-use
1	u:=a - b		
2	v:=c - a		
3	w:=u + v		
4	x:=d + b		
5	y:=c + 1	y,c:live	y:(6); c:none
6	z:=x * y	z live; x,y:dead	z:(7), x,y:none
7	d:=w - z	d live; w,z:dead	d, w,z: none

Var	Status	next-use
a	Live	None
b	Live	None
c	**Live**	**(5)**
d	Dead	None
u	Dead	None
v	Dead	None
w	Live	(7)
x	Live	(6)
y	**Dead**	**None**
z	Dead	None

Var	Status	next-use
a	Live	None
b	Live	(4)
c	Live	(5)
d	**Live**	**(4)**
u	Dead	None
v	Dead	None
w	Live	(7)
x	**Dead**	**None**
y	Dead	None
z	Dead	None

No	Instr	Status	next-use
1	u:=a - b		
2	v:=c - a		
3	w:=u + v		
4	x:=d + b	x,b:live;d:dead	y:(6):none
5	y:=c + 1	y,c:live	y:(6); c:none
6	z:=x * y	z live; x,y:dead	z:(7), x,y:none
7	d:=w - z	d live; w,z:dead	d, w,z: none

Final Table:

No	Instr	Status	next-use
1	u:=a - b	u, a, b: live	u:3; a:2; b: 4
2	v:=c - a	v,c, a: live	v:3; c:(5); a:none
3	w:=u + v	w live; u, v dead	w:(7), u, v: none
4	x:=d + b	x,b:live;d:dead	y:(6):none
5	y:=c + 1	y,c:live	y:(6); c:none
6	z:=x * y	z live; x,y:dead	z:(7), x,y:none
7	d:=w - z	d live; w,z:dead	d, w,z: none

Step 3: Allocation of registers and code generation

- Approach:
  - Look at each IR instruction and find corresponding set of machine instructions.
  - Check if variables in instruction are already in register. If so, use them otherwise load from memory.
  - Hold values of variables as long as possible. Store them back into location when running out of registers, jumping to a procedure or labelled statement. (Here data is not held in registers across block boundaries.)
- Generation of code for `a:=b + c`: Many possibilities with respect to where `a`, `b`, and `c` are stored, and how they will be used.
- The code generation algorithm keeps track of register contents and addresses for names:
  1. *Register Descriptor*: Keep track of what is currently in each register. As code generation progresses, each register will hold the value of names.
  2. *Address Descriptor*: Keep track of location where current value of a name can be found.
     Used to determine the accessing method for a name.

How to allocate registers

- GetReg: find a register for an operand y.

```
if     (y is in register R) and
       (R does not hold any other variables) and
       (y is dead and has no next use) then
       return R.
else if (there is an un-used register R) then
       return R
else
       Select an occupied register R for use
       Generate a Store instruction to
          save R's contents
       Update the descriptors and return R.
```

- Note:
  - GetReg consults an address table to find status of a variable
  - `a := b` can cause two variables to share the same register.
  - When saving some occupied register, find the one whose contents will be accessed as far possible down the block.

Final Code Generation

- Generate code for instruction of form `a:=b op c`:
  1. Check address table to determine where `b` is. If `b` is not in memory then use `GetReg` for a register `Rb` for `b`.
     If `GetReg` returns an empty register, generate:
     ```
     lw    Rb, offsetb($fp)
     ```
     Here, offset defines the offset of symbol `b` in the activation record.
     Also, assume that `b` is locally defined.
     Mark `b` to be located in `Rb`.
  2. Do the same for `c`. Note that you want to ensure that getting a register for `c` should not mean getting `Rb`. `GetReg` should be modified by making `Rb` unavailable.
     Generate if `c` is not in register:
     ```
     lw    Rc, offsetc($fp)
     ```
  3. Find a unique register `Ra` for holding `a`:
     ```
     op    Ra, Rc, Rb
     ```
  4. Mark that `a` is located in `Ra`.
- Example for statement `d:=(a - b) + (c - a) - (d + b)*(c+1)`

```
u:=a -b
v:=c - a
w:=u + v
x:=d + b
y:=c+1;
z:=x * y;
e:=w - z
```

Register allocation and code generation for code

- Initial table states: Registers

0, 1,12-15	Res.
2-11	Free

Variables

a-d	Memory
u-z	Nowhere

- For code (1) `u:=a - b`, Call GetReg to hold a, to hold b, and to hold u. Generate code accordingly:

Registers

0, 1	Res.
2	a
3	b
4	u
5-11	Free
12-15	Res.

Variables

a	2
b	3
u	4
c-d	Memory
v-z	Nowhere

Code:

```
lw $2, a
lw $3, b
sub $4, $2, $3
```

- For code (2) `v:=c - a`, Call GetReg for a, for c, and to hold v. Generate code accordingly:

Registers

0, 1,12-15	Res.
2	a
3	b
4	u
5	c
6	v
7-11	Free

Variables

a	2
b	3
u	4
c	5
v	6
d	Memory
w-z	Nowhere

Code:

```
lw $5, c
sub $6, $5, $2
```

- For code (3) `w:=u + v`, Call GetReg for u, for v, and to hold w. Note that u is dead (not used after this) Implies can use u's register.

Registers

0, 1, 12-15	Res.
2	a
3	b
4	w
5	c
6	v
7-11	Free

Variables

a	2
b	3
w	4
c	5
v	6
d	Memory
u, y-z	Nowhere

Code:

```
add $4, $4, $6
```

Register allocation and code generation for code - cont'd.

- For code `(4) x:=d + b`, Call GetReg for d, for b, and to hold x. v is not used again, so its register can be reused. GetReg will return that.

Registers

0, 1, 12-15	Res.
2	a
3	b
4	w
5	c
6	d
7	w
8	x
9-11	Free

Variables

a	2
b	3
d	6
c	5
w	7
x	8
d, u, v-z	Nowhere

Code:

```
lw $4, d
add $8, $4, $3
```

- For `(5) y:= c + 1`:

Registers

0, 1, 12-15	Res.
2	a
3	b
4	w
5	c
6	y
7	w
8	x
9-11	Free

Variables

a	2
b	3
y	6
c	5
w	7
x	8
d, u, v-z	Nowhere

Code:

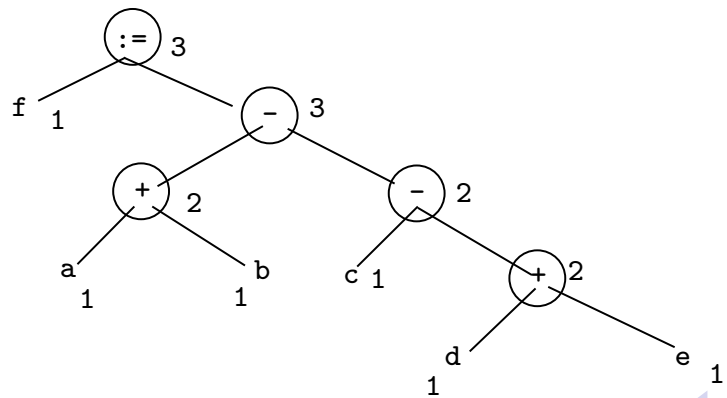
```
addi $6, $5, 1
```

Instruction Scheduling for Optimal Register Usage

- Earlier approach: Traverse left-to-right, bottom-to-top. Not the best choice if we want to minimize register-usage.
- A better algorithm: Select the side which has maximum register usage to generate code first.
  Two phases: i) Labeling phase ii) Code generation phase
- Labeling phase: Label each node of tree by their register requirements. Say we have a node op(t_1, t_2):
  1. if t_1 and t_2 are leaves: label both as 1 since they both need to be in a register.

     `regreq(`t_1`) = regreq(`t_2`) = 1;`
  2. Otherwise:
     Let r_1 = regreq(t_1), r_2 = regreq(t_2)
     regreq(op(t_1, t_2)) = $\max(r_1, r_2)$ if $r_1 \neq r_2$
     else $r_1 + 1$

- Example:



Code Generation Phase

- For a node op(t1, t2), generate code for t1, t2 and then for the node. Order of t1 and t2 depends on their register requirements.
- Manage registers by a stack: RSTACK. Stack initialized with full set of available registers. Operations on Stack: pop, push, top, exchange (swap top two registers)
- Notation: (t, r) to denote a node t with register requirement r.
- Convention: Topmost register at beginning of processing a subtree: result register for t.
- Algorithm:

```
GenCode(t) =
   if t is a leaf then
      generate ‘‘lw top(RSTACK), a’’
   else if t = op((t1, r1), (t2, r2)) then
      case r1 < r2:
         GenCode(t2);
         R:=pop(RSTACK);
         GenCode(t1);
         generate ‘‘op R, R, top(RSTACK)’’;
         push(RSTACK, R);
      case r1 >= r2:
         GenCode(t1);
         R:=pop(RSTACK);
         GenCode(t2);
         generate ‘‘op R, R, top(RSTACK)’’;
         push(RSTACK, R);
```