Compilers (CS31003)

Lecture 28-29

Target Code Generation

Target Code Generation – Scope

- Target Machine: `x86-32 bits`
- Input
  - Symbol Tables
  - Table of Labels
  - Table of Constants
  - Quad Array of TAC
- Output
  - List of Assembly Instructions
  - External Symbol Table and Link Information
- No Error / Exception Handling

TCG - tasks

- **Instruction Selection**
  - Level of the IR
    - High level IR ---- code templates
    - Low level IR --- direct translation
  - Nature of the instruction set architecture
  - Desired quality of the generated code

x = y + z

```
LD R0, y          // load y into register R0
ADD R0, R0, z   // add z to R0
ST  x, R0      // store R0 into x
```

Now,

```
a = b + c
d = a + e
a = a + 1      // INC a
```

- **Register Allocation and Assignment**
  - Allocation
  - Assignment

- **Instruction Ordering**

Target Language

- x = y – z
- b = a[i]
- a[j]=c
- x=*p
- *p=y
- if x < y goto L

```
LD R1, y
LD R2, z
SUB R1, R1, R2
ST x, R1
```

```
LD R1, i
MUL R1, R1,  8
LD R2, a(R1)
ST b, R2
```

```
LD R1, c
LD R2, j
MUL R2, R2, 8
ST a(R2), R1
```

```
LD R1, p
LD R2, 0(p)
ST x, R2
```

```
LD R1, p
LD R2, y
ST 0(R1), R2
```

```
LD R1, x
LD R2, y
SUB R1, R1, R2
BLTZ R1, M
```

- **Cost of a program**
  - *Loading*
    - Register-to-Register
    - Register-to-Memory and vice-versa
    - Indirect address
  - *Computation*

Target Code Generation Steps - Summary

[1] TAC Optimization

[2] Memory Binding

- Generate AR from ST – memory binding for local variables
- Generate Static Allocation from ST.gbl – memory binding for global variables
- Generate Constants from Table of Constants
- Register Allocations & Assignment

[3] Code Translation

- Generate Function Prologue
- Generate Function Epilogue
- Map TAC to Assembly – Function Body

[4] Target Code Optimization

[5] Target Code Management

- Integration into an Assembly File
- Link Information Generation – for multi-source build

TC Generation Steps – TAC Optimization: Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
  - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)
- Optimizations may be classified as *local* and *global*

TC Generation Steps – TAC Optimization

- Optimize TAC
- Peep-hole Optimization
  - Elimination of Useless Temporary
  - Eliminating Unreachable Code
  - Flow of Control Optimization
  - Algebraic Simplification & Reduction of Strength
- Common Sub-expression Elimination
- Constant Folding
- Dead-code Elimination

Basic Blocks

- **Definition:**

The maximal sequences of consecutive three-address instructions with the properties

1. The flow of control can enter only through first instruction. No jump to middle of the basic block is allowed.

2. Control will leave the block without halting or branching, except possibly at the last instruction in the block.

Effect of interrupts?? Like division by zero.

Basic Blocks

Algorithm: Partition IR (three-address instructions) into basic blocks

- **Input:** A sequence of IR (three-address instructions)
- **Output:** A list of basic blocks in which each instruction is assigned to exactly one basic block

- **Method:**

  Identify the leaders using following rules

  1. First three address code in the IR is a leader.
  2. Any instruction that is the target of the condition/unconditional jump is a leader.
  3. Any instruction that immediately follows a condition/unconditional jump is a leader.

  For each leader, its basic block consists of itself and all the instructions up to but not including the next leader or the end of IR.

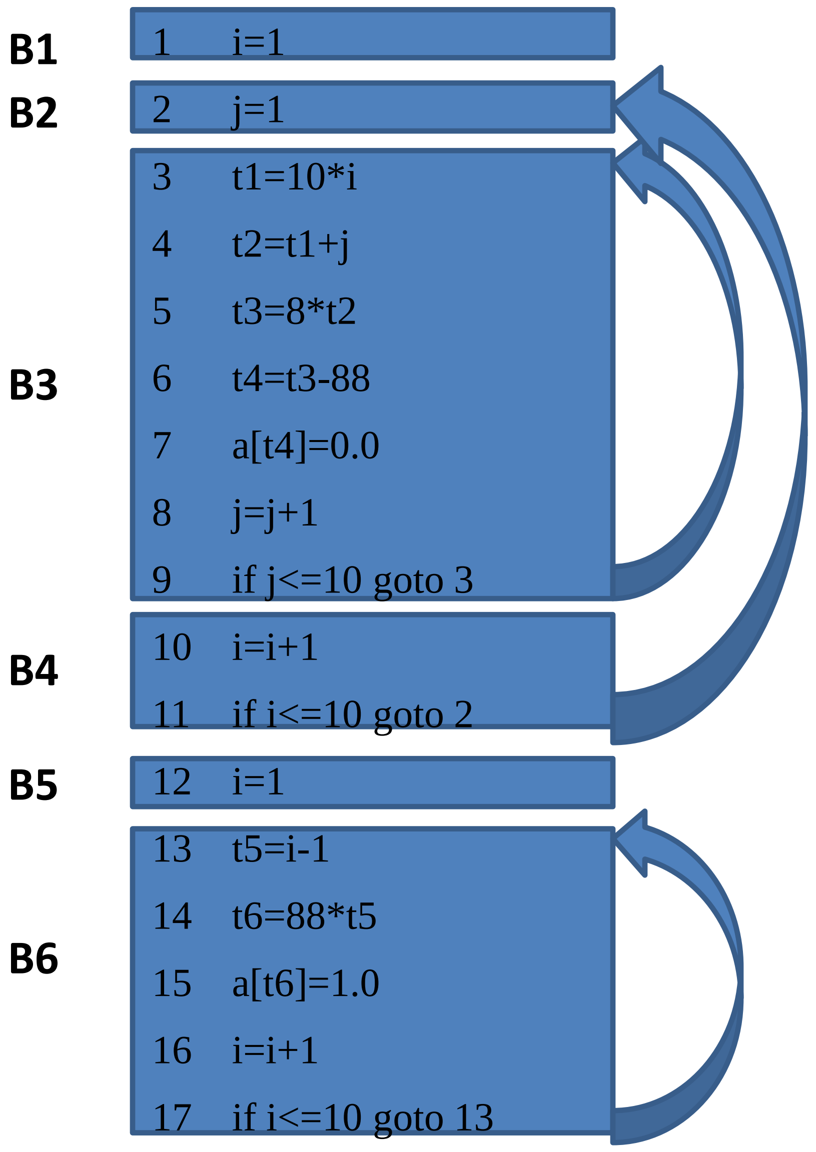
Basic Blocks

```
1    i=1
2    j=1
3    t1=10*i
4    t2=t1+j
5    t3=8*t2
6    t4=t3-88
7    a[t4]=0.0
8    j=j+1
9    if j<=10 goto 3
10   i=i+1
11   if i<=10 goto 2
12   i=1
13   t5=i-1
14   t6=88*t5
15   a[t6]=1.0
16   i=i+1
17   if i<=10 goto 13
```

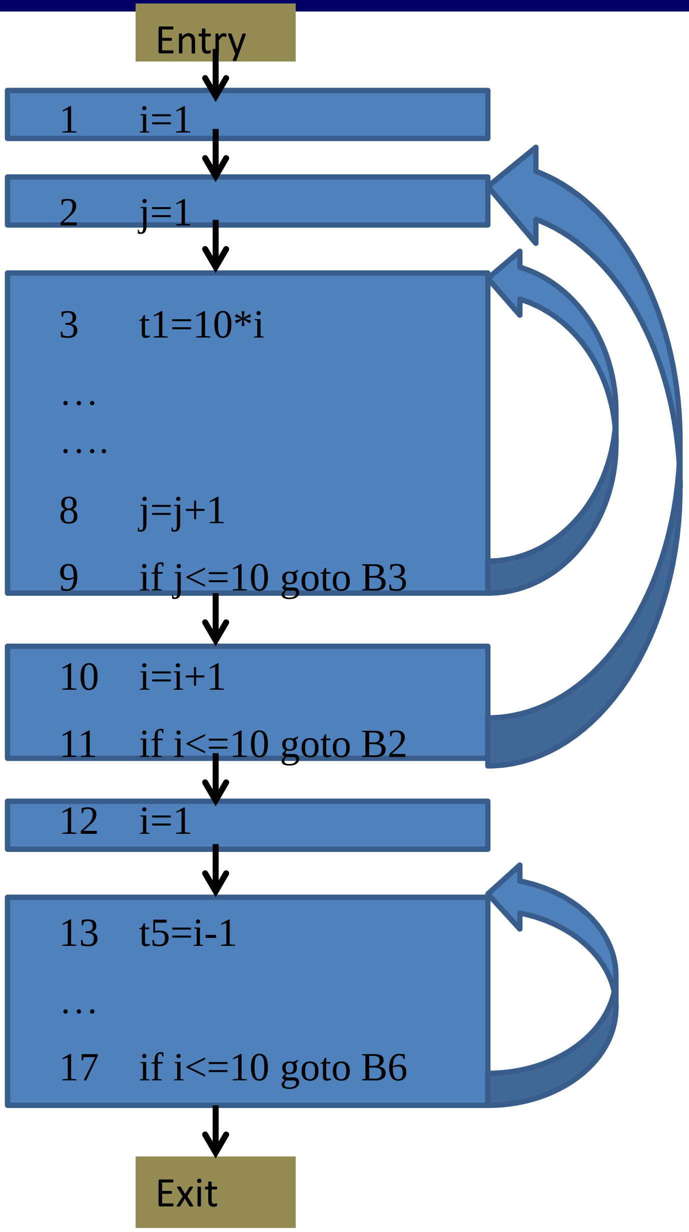
Basic Blocks

B1	1	i=1
B2	2	j=1
B3	3	t1=10*i
	4	t2=t1+j
	5	t3=8*t2
	6	t4=t3-88
	7	a[t4]=0.0
	8	j=j+1
	9	if j<=10 goto 3
B4	10	i=i+1
	11	if i<=10 goto 2
B5	12	i=1
B6	13	t5=i-1
	14	t6=88*t5
	15	a[t6]=1.0
	16	i=i+1
	17	if i<=10 goto 13

Basic Blocks to Flow Graphs



```
B1    1    i=1
B2    2    j=1
      3    t1=10*i
      4    t2=t1+j
      5    t3=8*t2
B3    6    t4=t3-88
      7    a[t4]=0.0
      8    j=j+1
      9    if j<=10 goto 3
B4    10   i=i+1
      11   if i<=10 goto 2
B5    12   i=1
      13   t5=i-1
      14   t6=88*t5
B6    15   a[t6]=1.0
      16   i=i+1
      17   if i<=10 goto 13
```



Loops???

1. B3
2. B6
3. {B2,B3,B4}

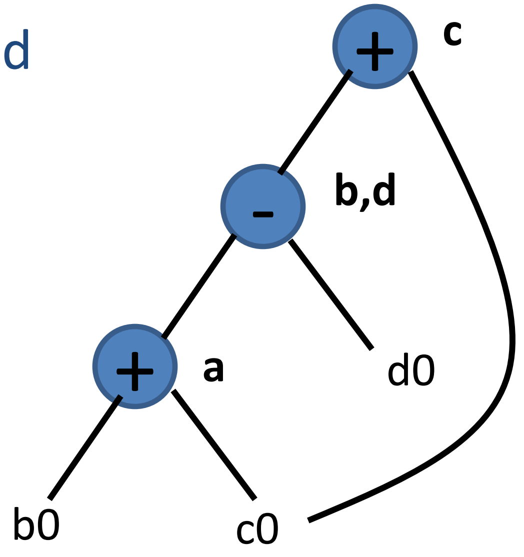
Optimization of Basic Blocks

- DAG representation
- Status of a variable: Live, Dead, Live on exit

Optimization of Basic Blocks

- Local Common Subexpression

```
a = b + c
b = a - d
c = b + c
d = a - d
```

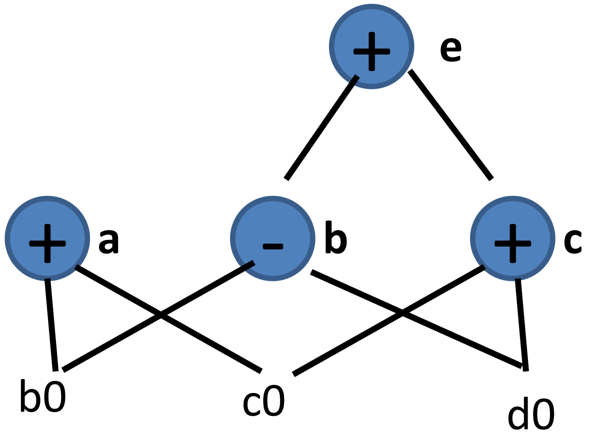


Is *b* live on exit?

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

e=b+c=(b-d)+(c+d)

Can you see?

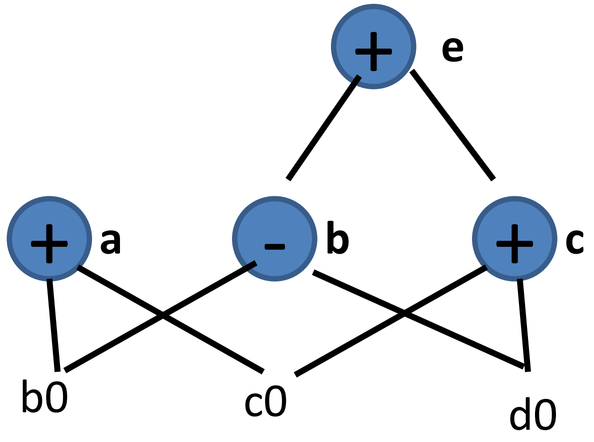


Optimization of Basic Blocks

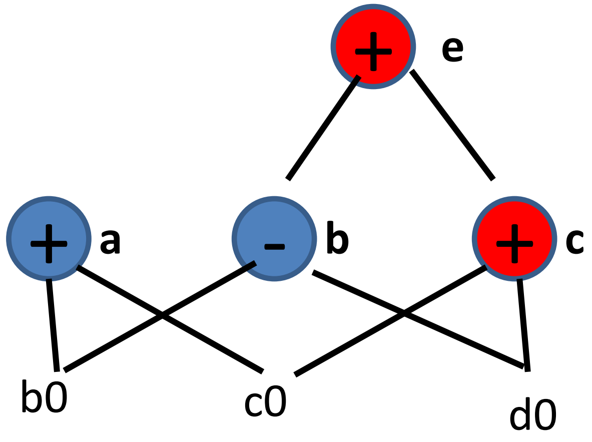
- Dead code elimination:

  Repeatedly delete root from a DAG that has no live variable attached.

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



e and c are not live on exit



Optimization of Basic Blocks

- Algebraic Identity and expression

Identity

$X + 0 = 0 + X = X$
$X - 0 = X$

$X \times 1 = 1 \times X = X$
$X / 1 = X$

Strength Reduction

$X^2 = X \times X$
$2 \times X = X + X$

Constant Folding

$2 * 3.14 / 1.21$

Generate IR

d = 2 * 3.14
a = b + c
e = c + d + b
f = 2 * 3.14 * a + e

Example: Vector Product

```
int a[5], b[5], c[5];
int i, n = 5;

for(i = 0; i < n; i++) {
    if (a[i] < b[i])
        c[i] = a[i] * b[i];
    else
        c[i] = 0;
}
return;
```

```
// int i, n = 5;
100: t1 = 5
101: n = t1
// for(i = 0; i < n; i++) {
102: t2 = 0
103: i = t2
104: if i < n goto 109 // T
105: goto 129 // F
106: t3 = i
107: i = i + 1
108: goto 104
// if (a[i] < b[i])
109: t4 = 4 * i
110: t5 = a[t4]
111: t6 = 4 * i
112: t7 = b[t6]
113: if t5 < t7 goto 115 // T
114: goto 124 // F
```

```
// c[i] = a[i] * b[i];
115: t8 = 4 * i
116: t9 = c + t8
117: t10 = 4 * i
118: t11 = a[t10]
119: t12 = 4 * i
120: t13 = b[t12]
121: t14 = t11 * t13
122: *t9 = t14
123: goto 106 // next
// c[i] = 0;
124: t15 = 4 * i
125: t16 = c + t15
126: t17 = 0
127: *t16 = t17
// }
128: goto 106 // for
// return;
129: return
```

Example: Vector Product: Peep-hole Optimization

Peep-hole optimization and potential removals are marked. Recomputed quad numbers are shown:

```
    // int i, n = 5;
    100: t1 = 5 <=== def-use propagation: XXX
100:101: n = 5
    // for(i = 0; i < n; i++) {
    102: t2 = 0 <=== def-use propagation: XXX
101:103: i = 0
102:104: if i < n goto 109 // true exit
103:105: goto 129 // false exit
    106: t3 = i <=== unused: XXX
104:107: i = i + 1
105:108: goto 104
    // if (a[i] < b[i])
106:109: t4 = 4 * i // strength reduction
107:110: t5 = a[t4]
108:111: t6 = 4 * i // strength reduction
109:112: t7 = b[t6]
110:113: if t5 >= t7 goto 124
    114: goto 115 <=== jmp-to-fall through: XXX
```

```
    // c[i] = a[i] * b[i];
111:115: t8 = 4 * i // strength reduction
112:116: t9 = c + t8
113:117: t10 = 4 * i // strength reduction
114:118: t11 = a[t10]
115:119: t12 = 4 * i // strength reduction
116:120: t13 = b[t12]
117:121: t14 = t11 * t13
118:122: *t9 = t14
119:123: goto 106 // next exit
    // c[i] = 0;
120:124: t15 = 4 * i // strength reduction
121:125: t16 = c + t15
    126: t17 = 0 <=== def-use propagation: XXX
122:127: *t16 = 0
    // } // End of for loop
123:128: goto 106
    // return;
124:129: return
```

Example: Vector Product: Peep-hole Optimization: Common Sub-Expression (CSE)

On removal, strength reduction, and compaction:

```
100: n = 5
101: i = 0
102: if i < n goto 106
103: goto 124
104: i = i + 1
105: goto 102
106: t4 = i << 2 // CSE
107: t5 = a[t4]
108: t6 = i << 2 // CSE
109: t7 = b[t6]
110: if t5 >= t7 goto 120
```

```
111: t8 = i << 2 // CSE
112: t9 = c + t8
113: t10 = i << 2 // CSE
114: t11 = a[t10]
115: t12 = i << 2 // CSE
116: t13 = b[t12]
117: t14 = t11 * t13
118: *t9 = t14
119: goto 104
120: t15 = i << 2 // CSE
121: t16 = c + t15
122: *t16 = 0
123: goto 104
124: return
```

Replace `4 * i` with `i << 2`

Example: Vector Product: Common Sub-Expression (CSE): Elimination

Substitute `i << 2` by `t4`:

```
100: n = 5
101: i = 0
102: if i < n goto 106
103: goto 124
104: i = i + 1
105: goto 102
106: t4 = i << 2 // CSE
107: t5 = a[t4]
108: t6 = t4 // CSE
109: t7 = b[t6]
110: if t5 >= t7 goto 120
```

```
111: t8 = t4 // CSE
112: t9 = c + t8
113: t10 = t4 // CSE
114: t11 = a[t10]
115: t12 = t4 // CSE
116: t13 = b[t12]
117: t14 = t11 * t13
118: *t9 = t14
119: goto 104
120: t15 = t4 // CSE
121: t16 = c + t15
122: *t16 = 0
123: goto 104
124: return
```

Since `i` changes only at 104; `t4`, once computed, does not change during the iteration (How do we know?)

Example: Vector Product: Copy Propagation

CSE generates several single variable copies. We can propate them - push them down

```
100: n = 5
101: i = 0
102: if i < n goto 106
103: goto 124
104: i = i + 1
105: goto 102
106: t4 = i << 2
107: t5 = a[t4]
108: t6 = t4
109: t7 = b[t4] // Copy Propagation
110: if t5 >= t7 goto 120
```

```
111: t8 = t4
112: t9 = c + t4 // Copy Propagation
113: t10 = t4
114: t11 = a[t4] // Copy Propagation
115: t12 = t4
116: t13 = b[t4] // Copy Propagation
117: t14 = t11 * t13
118: *t9 = t14
119: goto 104
120: t15 = t4
121: t16 = c + t4 // Copy Propagation
122: *t16 = 0
123: goto 104
124: return
```

`t6`, `t8`, `t10`, `t12`, and `t15` are all copies of `t4`

Example: Vector Product: Deadcode Elimination & CSE

As copies are propagated, the assignments to the earlier variables become useless - called Deadcode

```
100: n = 5
101: i = 0
102: if i < n goto 106
103: goto 124
104: i = i + 1
105: goto 102
106: t4 = i << 2
107: t5 = a[t4]
108: t6 = t4  // Deadcode
109: t7 = b[t4]
110: if t5 >= t7 goto 120
```

```
111: t8 = t4  // Deadcode
112: t9 = c + t4
113: t10 = t4  // Deadcode
114: t11 = a[t4]
115: t12 = t4  // Deadcode
116: t13 = b[t4]
117: t14 = t11 * t13
118: *t9 = t14
119: goto 104
120: t15 = t4  // Deadcode
121: t16 = c + t4
122: *t16 = 0
123: goto 104
124: return
```

The deadcode does not contribute to the computation. They can be removed

Example: Vector Product: Deadcode Elimination and more CSE

We just erase those dead quads

```
100: n = 5
101: i = 0
102: if i < n goto 106
103: goto 124
104: i = i + 1
105: goto 102
106: t4 = i << 2
107: t5 = a[t4]       // CSE
108:                  // Deadcode eliminated
109: t7 = b[t4]       // CSE
110: if t5 >= t7 goto 120
```

```
111:                  // Deadcode eliminated
112: t9 = c + t4
113:                  // Deadcode eliminated
114: t11 = a[t4]      // CSE
115:                  // Deadcode eliminated
116: t13 = b[t4]      // CSE
117: t14 = t11 * t13
118: *t9 = t14
119: goto 104
120:                  // Deadcode eliminated
121: t16 = c + t4
122: *t16 = 0
123: goto 104
124: return
```

There are two array expressions that are common and can be eliminated

Example: Vector Product: CSE, Copy Propagation & Constant Folding

On CSE, we can propagate the copies

```
100: n = 5
101: i = 0
102: if i < 5 goto 106 // Const. Fold.
103: goto 124
104: i = i + 1
105: goto 102
106: t4 = i << 2
107: t5 = a[t4]       // CSE
108:
109: t7 = b[t4]       // CSE
110: if t5 >= t7 goto 120
```

```
111:
112: t9 = c + t4
113:
114: t11 = t5    // CSE
115:
116: t13 = t7    // CSE
117: t14 = t5 * t7 // Copy Propagation
118: *t9 = t14
119: goto 104
120:
121: t16 = c + t4
122: *t16 = 0
123: goto 104
124: return
```

We also fold the constant (`n`)

Example: Vector Product: More Deadcode

This creates more dead quads

```
100: n = 5   // Deadcode
101: i = 0
102: if i < 5 goto 106
103: goto 124
104: i = i + 1
105: goto 102
106: t4 = i << 2
107: t5 = a[t4]
108:
109: t7 = b[t4]
110: if t5 >= t7 goto 120
111:
112: t9 = c + t4
113:
114: t11 = t5   // Deadcode
115:
116: t13 = t7   // Deadcode
117: t14 = t5 * t7
118: *t9 = t14
119: goto 104
120:
121: t16 = c + t4
122: *t16 = 0
123: goto 104
124: return
```

Example: Vector Product: Deadcode Elimination

On elimination

```
100:             // Deadcode
101: i = 0
102: if i < 5 goto 106
103: goto 124
104: i = i + 1
105: goto 102
106: t4 = i << 2
107: t5 = a[t4]
108:
109: t7 = b[t4]
110: if t5 >= t7 goto 120
```

```
111:
112: t9 = c + t4
113:
114:             // Deadcode
115:
116:             // Deadcode
117: t14 = t5 * t7
118: *t9 = t14
119: goto 104
120:
121: t16 = c + t4
122: *t16 = 0
123: goto 104
124: return
```

Example: Vector Product: Compacted Code and Advanced Optimizations

```
100:101: i = 0
101:102: if i < 5 goto 105:106
102:103: goto 116:124
103:104: i = i + 1
104:105: goto 101:102
105:106: t4 = i << 2
106:107: t5 = a[t4]
107:109: t7 = b[t4]
108:110: if t5 >= t7 goto 113:120
109:112: t9 = c + t4
110:117: t14 = t5 * t7
111:118: *t9 = t14
112:119: goto 103:104
113:121: t16 = c + t4
114:122: *t16 = 0
115:123: goto 103:104
116:124: return
```

```
100:101: i = 0                 // t4 = 0
101:102: if i < 5 goto 105:106 // t4 < 20
102:103: goto 116:124
103:104: i = i + 1             // Where is it used?
104:105: goto 101:102
105:106: t4 = i << 2           // t4 = t4 + 4. t4 == 4 * i
106:107: t5 = a[t4]
107:109: t7 = b[t4]
108:110: if t5 >= t7 goto 113:120
109:112: t9 = c + t4           // CSE ?
110:117: t14 = t5 * t7
111:118: *t9 = t14
112:119: goto 103:104
113:121: t16 = c + t4          // CSE ?
114:122: *t16 = 0
115:123: goto 103:104
116:124: return
```

The above marked optimizations need:

- **Computation of Loop Invariant**: Note that `i` and `t4` change in sync always (on all paths) with `t4 = 4 * i` and `i` is used only to compute `t4` in every iteration. So we can change the loop control from `i` to `t4` directly and eliminate `i`
- **Code Movement**: Code for `c[i]` is common on both true and false paths of the condition check as `c + t4`. It can be moved before the condition check and one of them can be eliminated.

TC Generation Steps – Memory Binding

- Generate AR from ST – memory binding for local variables

```
int Sum(int a[], int n) {
    int i, s = 0;
    for(i = 0; i < n; ++i) {
        int t;
        t = a[i];
        s += t;
    }
    return s;
}
```

```
Sum:    s = 0
        i = 0
L0:     if i < n goto L2
        goto L3
L1:     i = i + 1
        goto L0
L2:     t1 = i * 4
        t_1 = a[t1]
        s = s + t_1
        goto L1
L3:     return s
```

Symbol Table

a	int[]	param	4	0
n	int	param	4	4
i	int	local	4	8
s	int	local	4	12
t_1	int	local	4	16
t1	int	temp	4	20

Activation Record

t1	int	temp	4	–16
t_1	int	local	4	–12
s	int	local	4	–8
i	int	local	4	–4
a	int[]	param	4	+8
n	int	param	4	+12

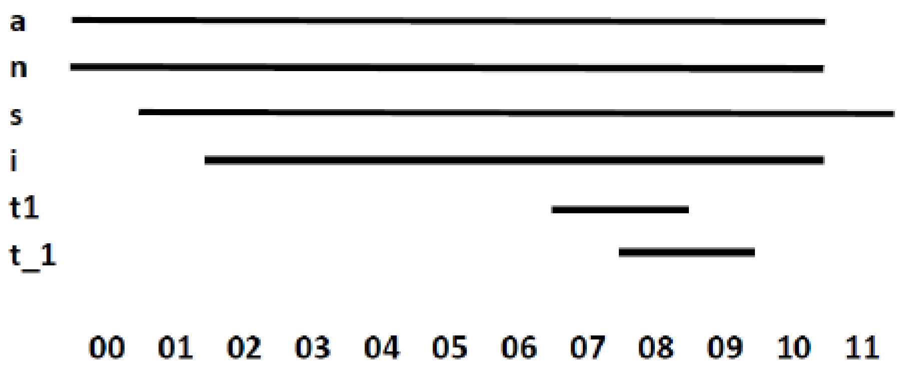
TC Generation Steps – Memory Binding

- Generate Static Allocation from ST.gbl – memory binding for global variables
  - Use DATA SEGMENT
- Generate Constants from Table of Constants
  - Use CONST SEGMENT
- Create memory binding for variables – register allocations
  - After a load / store the variable on the activation record and the register have identical values
  - Register allocations are often used to pass `int` or pointer parameters
  - Register allocations are often used to return `int` or pointer values

TC Generation Steps – Register Allocation & Assignment

- DEF-USE / Liveness Analysis / Interval Graph

```
000:                          // a, n
001:     s = 0                // a, n, s
002:     i = 0                // a, n, s, i
003: L0: if i < n goto L2    // a, n, s, i
004:     goto L3              // a, n, s, i
005: L1: i = i + 1            // a, n, s, i
006:     goto L0              // a, n, s, i
007: L2: t1 = i * 4           // a, n, s, i, t1
008:     t_1 = a[t1]          // a, n, s, i, t1, t_1
009:     s = s + t_1          // a, n, s, i, t_1
010:     goto L1              // a, n, s, i
011: L3: return s             // s
```



TC Generation Steps – Register Allocation & Assignment

Using a linear scan algorithm one can allocate and assign registers:

1 Perform DFA to gather liveness information. Keep track of all variables' live intervals, the interval when a variable is live, in a list sorted in order of increasing start point (this ordering is free if the list is built when computing liveness). We consider variables and their intervals to be interchangeable in this algorithm.

2 Iterate through liveness start points and allocate a register from the available register pool to each live variable.

TC Generation Steps – Register Allocation & Assignment

3 At each step maintain a list of active intervals sorted by the end point of the live intervals. (Note that insertion sort into a balanced binary tree can be used to maintain this list at linear cost). Remove any expired intervals from the active list and free the expired interval's register to the available register pool.

4 In the case where the active list is size R we cannot allocate a register. In this case add the current interval to the active pool without allocating a register. Spill the interval from the active list with the furthest end point. Assign the register from the spilled interval to the current interval or, if the current interval is the one spilled, do not change register assignments.

TC Generation Steps – Code Translation

- Generate Function Prologue – few lines of code at the beginning of a function, which prepare the stack and registers for use within the function
  - Pushes the old base pointer onto the stack, such that it can be restored later.

    ```
    push ebp
    ```
  - Assigns the value of stack pointer (which is pointed to the saved base pointer and the top of the old stack frame) into base pointer such that a new stack frame will be created on top of the old stack frame.

    ```
    mov ebp, esp
    ```
  - Moves the stack pointer further by decreasing its value to make room for variables (i.e. the function's local variables).

    ```
    sub esp, 12
    ```
  - Save the registers on the stack by push

    ```
    push esi
    ```

TC Generation Steps – Code Translation

- Generate Function Epilogue – appears at the end of the function, and restores the stack and registers to the state they were in before the function was called
  - Restore the registers from the stack by pop

    ```
    pop esi
    ```
  - Replaces the stack pointer with the current base (or frame) pointer, so the stack pointer is restored to its value before the prologue

    ```
    mov esp, ebp
    ```
  - Pops the base pointer off the stack, so it is restored to its value before the prologue

    ```
    pop ebp
    ```
  - Returns to the calling function, by popping the previous frame's program counter off the stack and jumping to it

    ```
    ret 0
    ```

TC Generation Steps – Code Translation

- Map TAC to Assembly
  - Choose optimized assembly instructions
  - Algebraic Simplification & Reduction of Strength
  - Use of Machine Idioms

TC Generation Steps – Target Code Optimization

- Optimize Target Code
  - Eliminating Redundant Load-Store
  - Eliminating Unreachable Code
  - Flow of Control Optimization

TC Generation Steps – Target Code Management

- Integration into an Assembly File
- Link Information Generation – for multi-source build

Code generator

- Some or all of the operands of an operator must be in registers
- Store temporaries in registers
- Registers are used to hold global values
- Registers are often used to help with run-time storage management

LD *reg*, *mem*

ST *mem*, *reg*

ADD *reg*, *reg*, *reg*

Code generation algorithm

- Registers and Address Descriptors

- Add Operation
  - x=y+z
  - If y is not in register
    - getReg(y) // LD R_y, *y*
  - If z is not in register
    - getReg(z) // LD R_z, *z*
  - ADD R_x, R_y, R_z
  - ST *x*,R_x

- Copy Operation
  - x=y
  - If y is not in register
    - getReg(y) // LD R_y, *y*
  - ST *x*,R_y

x = y + z

x= z + y

Ending the basic block?

Managing register and address descriptors

t = a – b	LD R1, a LD R2 b SUB R2, R1, R2
u = a – c	LD R3, c SUB R1, R1, R3
v = t + u	ADD R3, R2, R1
a = d	LD R2, d
d = v + u	ADD R1, R3, R1
	ST a, R2 ST d, R1

a, b, c, d live on exit.

Rest are temporaries to the block.

Design of *getReg*(***I***)

- Is I in register?
- If not, is there any register available?
- If not, *spill*

- Compute *score* (number of store instructions). Choose minimum *score* for *spill*.

Peephole Optimization

- Redundant instruction elimination
  - LD R0, a
  - ST a, R0

Is it always redundant?

- Algebraic simplification

- Machine Idioms

Peephole Optimization

- Flow of control

```
      if debug==1 goto L1
      goto L2
L1: print debugging information
L2:

      if debug!=1 goto L2
      print debugging information
```

```
      goto L1
      ……
L1:   goto L2
```

Register allocation and assignment

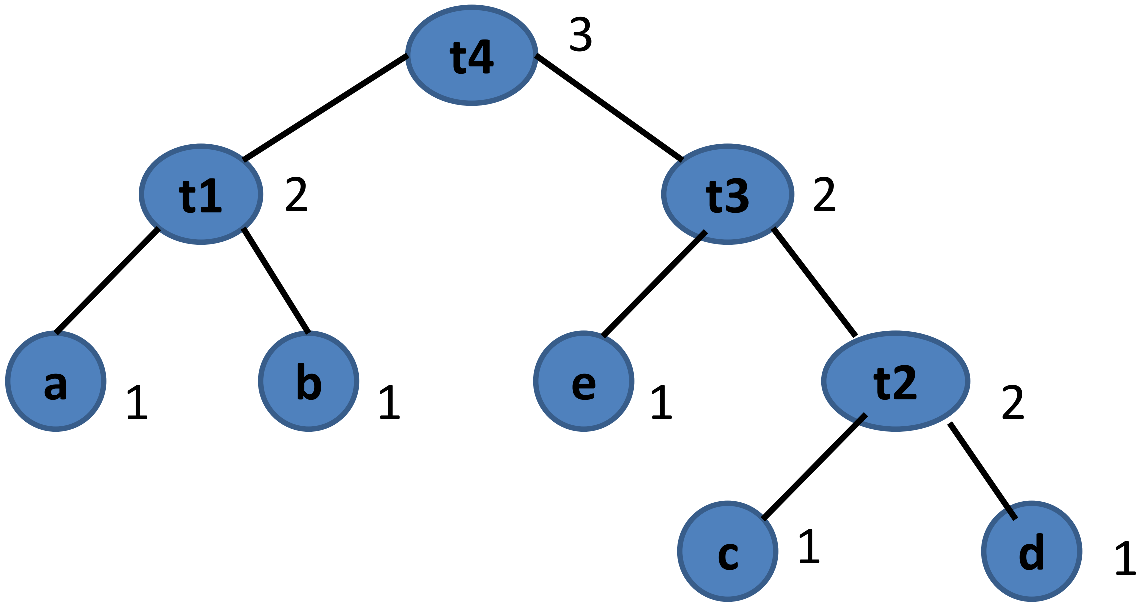
- $x = a + b \times c$
- $x = a / (b+c) - d \times (e+f)$

Similarity with graph coloring problem

Ershov Numbers

1. Label all leaves 1
2. The label of an interior node with one child is the label of its child.
3. The label of an interior node with two children is
   1. One plus the label of its children if the labels are same.
   2. Else the larger of the labels of its children

```
t1 = a – b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```



Code generation from labeled expression tree

```
t1 = a – b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```

- With R=3

```
LD    R3, d
LD    R2, c
ADD   R3, R2, R3
LD    R2, e
MUL   R3, R2, R3
LD    R2, b
LD    R1, a
SUB   R2, R1, R2
ADD   R3, R2, R3
```

Code generation from labeled expression tree

- With R=3

```
t1 = a – b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```

```
LD    R3, d
LD    R2, c
ADD   R3, R2, R3
LD    R2, e
MUL   R3, R2, R3
LD    R2, b
LD    R1, a
SUB   R2, R1, R2
ADD   R3, R2, R3
```

Workout: Start computation from t1.

Code generation

- With R=2

```
t1 = a – b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```

```
LD    R2, d
LD    R1, c
ADD   R2, R1, R2
LD    R1, e
MUL   R2, R1, R2
ST    t3, R2          spill
LD    R2, b
LD    R1, a
SUB   R2, R1, R2
LD    R1, t3
ADD   R3, R2, R3
```

Code Mapping

Code Mapping – Unary, Binary & Copy Assignment

int a, b, c;

TAC	x86	Remarks
`a = 5`	`mov DWORD PTR _a$[ebp], 5`	mov r/m32,imm32: Move imm32 to r/m32.

`a = b`	`mov eax, DWORD PTR _b$[ebp]`
	`mov DWORD PTR _a$[ebp], eax`
		mov r32,r/m32: Move r/m32 to r32.
		mov r/m32,r32: Move r32 to r/m32.

`a = -b`	`mov eax, DWORD PTR _b$[ebp]`
	`neg eax`
	`mov DWORD PTR _a$[ebp], eax`
		neg r/m32: Two's complement negate r/m32.

`a = b + c`	`mov eax, DWORD PTR _b$[ebp]`
	`add eax, DWORD PTR _c$[ebp]`
	`mov DWORD PTR _a$[ebp], eax`
		add r32, r/m32: Add r/m32 to r32

`a = b - c`	`mov eax, DWORD PTR _b$[ebp]`
	`sub eax, DWORD PTR _c$[ebp]`
	`mov DWORD PTR _a$[ebp], eax`
		sub r32,r/m32: Subtract r/m32 from r32.

`a = b * c`	`mov eax, DWORD PTR _b$[ebp]`
	`imul eax, DWORD PTR _c$[ebp]`
	`mov DWORD PTR _a$[ebp], eax`
		imul r/m32: EDX:EAX = EAX * r/m doubleword.

`a = b / c`	`mov eax, DWORD PTR _b$[ebp]`
	`cdq`
	`idiv DWORD PTR _c$[ebp]`
	`mov DWORD PTR _a$[ebp], eax`
		cdq: EDX:EAX = sign-extend of EAX. Convert Doubleword to Quadword
		idiv r/m32: Signed divide EDX:EAX by r/m32, with result stored in EAX = Quotient, EDX = Remainder.

`a = b % c`	`mov eax, DWORD PTR _b$[ebp]`
	`cdq`
	`idiv DWORD PTR _c$[ebp]`
	`mov DWORD PTR _a$[ebp], edx`	

Code Mapping – Unconditional & Conditional Jump

TAC	x86	Remarks
`goto L1`	`jmp  SHORT $L1$1017`	jmp rel8: Jump short, relative, displacement relative to next instruction. Mapped target address for L1 is $L1$1017.

`if a < b goto L1`	`mov eax, DWORD PTR _a$[ebp]`
	`cmp eax, DWORD PTR _b$[ebp]`
	`jge SHORT $LN1@main`
	`jmp SHORT $L1$1018`
	`$LN1@main:`
		cmp r32,r/m32: Compare r/m32 with r32. Compares the first operand with the second operand and sets the status flags in the EFLAGS register according to the results.
		jge rel8: Jump short if greater or equal (SF=OF).
		Input label L1 transcoded to $L1$1018 and new temporary label $LN1@main used.

`if a == b goto L1`	`mov  eax, DWORD PTR _a$[ebp]`
	`cmp  eax, DWORD PTR _b$[ebp]`
	`jne  SHORT $LN1@main`
	`jmp  SHORT $L1$1018`
	`$LN1@main:`
		jne rel8: Jump short if not equal (ZF=0).

`if a goto L1`	`cmp  DWORD PTR _a$[ebp], 0`
	`je  SHORT $LN1@main`
	`jmp  SHORT $L1$1018`
	`$LN1@main:`
		je rel8: Jump short if equal (ZF=1).

`ifFalse a goto L1`	`cmp  DWORD PTR _a$[ebp], 0`
	`jne  SHORT $LN1@main`
	`jmp  SHORT $L1$1018`
	`$LN1@main:`	

Code Mapping – Function Call & Return

```
int f(int x, int y, int z) { int m = 5; return m; }
...
int a, b, c, d;
d = f(a, b, c);
```

TAC	x86	Remarks

`param a`
`param b`
`param c`
`d = call f, 3`
	`mov eax, DWORD PTR _c$[ebp]`
	`push eax`
	`mov eax, DWORD PTR _b$[ebp]`
	`push eax`
	`mov eax, DWORD PTR _a$[ebp]`
	`push eax`
	`call _f`
		push r32: Push r32. Decrements the stack pointer and then stores the source operand on the top of the stack.
		call rel32: Call near, relative, displacement relative to next instruction. Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand.

	`add esp, 12 ; 0000000cH`	Adjust the stack pointer back (for parameters)
	`mov DWORD PTR _c$[ebp], eax`	Return value passed through eax

`In f()`	`push ebp`
	`mov ebp, esp`
		Save base pointer & set new base pointer

`return m`	`mov eax, DWORD PTR _m$[ebp]`
	`mov esp, ebp`
	`pop ebp`
	`ret 0`
		pop r/m32: Pop top of stack into m32; increment stack pointer.
		ret imm16: Near return to calling procedure and pop imm16 bytes from stack..

Code Mapping – Indexed Copy, Address & Pointer Assignment

int a, x[10], i = 0, b, *p = 0;

TAC	x86	Remarks

`a = x[i]`	`mov edx, DWORD PTR _i$[ebp]`
	`mov eax, DWORD PTR _x$[ebp+edx*4]`
	`mov DWORD PTR _a$[ebp], eax`	

`x[i] = b`	`mov edx, DWORD PTR _i$[ebp]`
	`mov eax, DWORD PTR _b$[ebp]`
	`mov DWORD PTR _x$[ebp+edx*4], eax`	

`p = &a`	`lea eax, DWORD PTR _a$[ebp]`
	`mov DWORD PTR _p$[ebp], eax`
		lea r32,m: Store effective address for m in register r32. Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register.

`a = *p`	`mov eax, DWORD PTR _p$[ebp]`
	`mov ecx, DWORD PTR [eax]`
	`mov DWORD PTR _a$[ebp], ecx`	

`*p = b`	`mov eax, DWORD PTR _p$[ebp]`
	`mov ecx, DWORD PTR _b$[ebp]`
	`mov DWORD PTR [eax], ecx`	

Code Mapping – Unary, Binary & Copy Assignment: `double`

double a = 1, b = 7, c = 2;
CONST SEGMENT
__real@40140000 DQ 040140000r ; 5
__real@401c0000 DQ 0401c0000r ; 7
__real@40000000 DQ 040000000r ; 2
__real@3ff00000 DQ 03ff00000r ; 1

TAC	x86	Remarks

`a = 5`	`fld QWORD PTR __real@40140000`
	`fstp QWORD PTR _a$[ebp]`
		fld m32fp: Push m32fp onto the FPU register stack.
		fstp m32fp: Copy ST(0) to m32fp and pop register stack.

`a = b`	`fld QWORD PTR _b$[ebp]`
	`fstp QWORD PTR _a$[ebp]`	

`a = -b`	`fld QWORD PTR _b$[ebp]`
	`fchs`
	`fstp QWORD PTR _a$[ebp]`
		fchs: Change Sign. Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa.

`a = b + c`	`fld QWORD PTR _b$[ebp]`
	`fadd QWORD PTR _c$[ebp]`
	`fstp QWORD PTR _a$[ebp]`
		fadd m32fp: Add m32fp to ST(0) and store result in ST(0).

`a = b - c`	`fld QWORD PTR _b$[ebp]`
	`fsub QWORD PTR _c$[ebp]`
	`fstp QWORD PTR _a$[ebp]`
		fsub m32fp: Subtract m32fp from ST(0) and store result in ST(0).

`a = b * c`	`fld QWORD PTR _b$[ebp]`
	`fmul QWORD PTR _c$[ebp]`
	`fstp QWORD PTR _a$[ebp]`
		fmul m32fp: Multiply ST(0) by m32fp and store result in ST(0).

`a = b / c`	`fld QWORD PTR _b$[ebp]`
	`fdiv QWORD PTR _c$[ebp]`
	`fstp QWORD PTR _a$[ebp]`
		fdiv m32fp: Divide ST(0) by m32fp and store result in ST(0).