6.035

Spring 2010

Loop Optimizations

Instruction Scheduling

Outline

- **Scheduling for loops**
- Loop unrolling
- Software pipelining
- Interaction with register allocation
- Hardware vs. Compiler
- Induction Variable Recognition
- loop invariant code motion

Scheduling Loops

- Loop bodies are small
- But, lot of time is spend in loops due to large number of iterations
- Need better ways to schedule loops

Loop Example

- Machine
  - One load/store unit
    - load 2 cycles
    - store 2 cycles
  - Two arithmetic units
    - add 2 cycles
    - branch 2 cycles
    - multiply 3 cycles
  - Both units are pipelined (initiate one op each cycle)
- Source Code

```
for i = 1 to N
    A[i] = A[i] * b
```

Loop Example

- Source Code

```
for i = 1 to N
   A[i] = A[i] * b
```

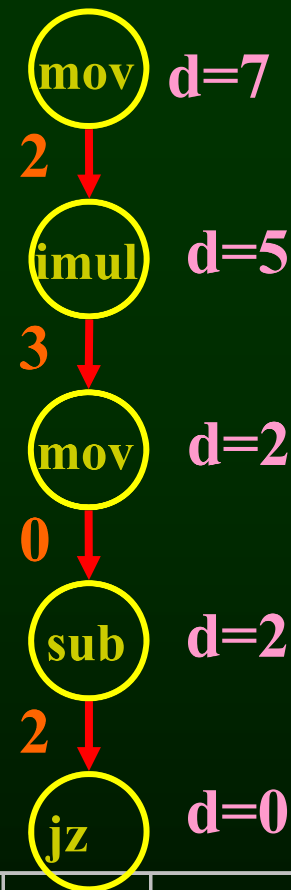
- Assembly Code

```
loop:
  mov   (%rdi,%rax), %r10
  imul  %r11, %r10
  mov   %r10, (%rdi,%rax)
  sub   $4, %rax
  jz    loop
```

base

offset

Loop Example



- Assembly Code

```
loop:
  mov    (%rdi,%rax), %r10
  imul   %r11, %r10
  mov    %r10, (%rdi,%rax)
  sub    $4, %rax
  jz     loop
```

- Schedule (9 cycles per iteration)

mov					mov							
	mov					mov						
		imul					bge					
			imul					bge				
				imul								
					sub							
						sub						

Outline

- Scheduling for loops
- **Loop unrolling**
- Software pipelining
- Interaction with register allocation
- Hardware vs. Compiler
- Induction Variable Recognition
- loop invariant code motion

Loop Unrolling

- Unroll the loop body few times
- Pros:
  - Create a much larger basic block for the body
  - Eliminate few loop bounds checks
- Cons:
  - Much larger program
  - Setup code (# of iterations < unroll factor)
  - beginning and end of the schedule can still have unused slots

Loop Example

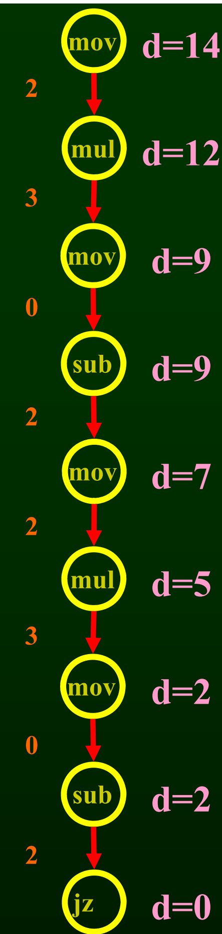
```
loop:
  mov    (%rdi,%rax), %r10
  imul   %r11, %r10
  mov    %r10, (%rdi,%rax)
  sub    $4, %rax
  jz     loop
```

Loop Example

```
loop:
  mov     (%rdi,%rax), %r10
  imul    %r11, %r10
  mov     %r10, (%rdi,%rax)
  sub     $4, %rax
  mov     (%rdi,%rax), %r10
  imul    %r11, %r10
  mov     %r10, (%rdi,%rax)
  sub     $4, %rax
  jz      loop
```

Loop Example

```
loop:
     mov     (%rdi,%rax), %r10
     imul    %r11, %r10
     mov     %r10, (%rdi,%rax)
     sub     $4, %rax
     mov     (%rdi,%rax), %r10
     imul    %r11, %r10
     mov     %r10, (%rdi,%rax)
     sub     $4, %rax
     jz      loop
```



- Schedule (8 cycles per iteration)

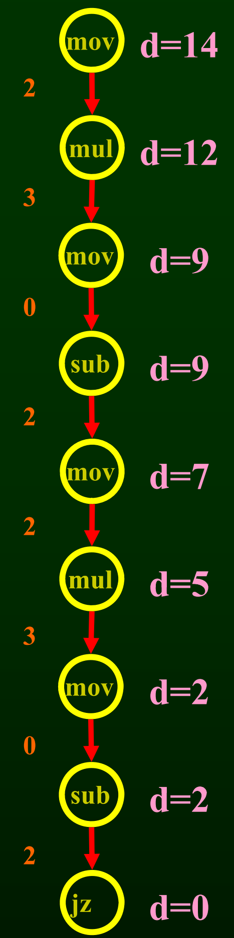
mov					mov		mov					mov			
	mov					mov		mov					mov		
		imul							imul					bge	
			imul							imul					bge
				imul							imul				
					sub							sub			
						sub							sub		

Loop Unrolling

- Rename registers
  - Use different registers in different iterations

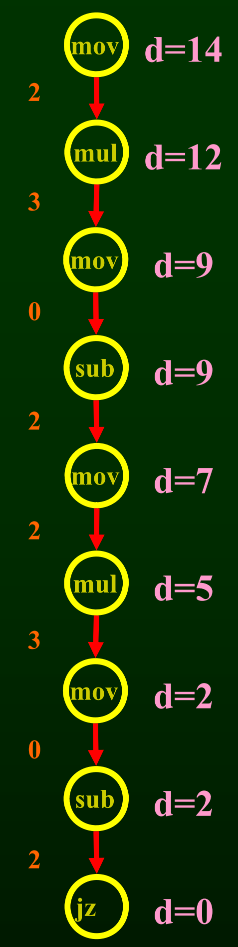
Loop Example

```
loop:
    mov    (%rdi,%rax), %r10
    imul   %r11, %r10
    mov    %r10, (%rdi,%rax)
    sub    $4, %rax
    mov    (%rdi,%rax), %r10
    imul   %r11, %r10
    mov    %r10, (%rdi,%rax)
    sub    $4, %rax
    jz     loop
```



Loop Example

```
loop:
    mov    (%rdi,%rax), %r10
    imul   %r11, %r10
    mov    %r10, (%rdi,%rax)
    sub    $4, %rax
    mov    (%rdi,%rax), %rcx
    imul   %r11, %rcx
    mov    %rcx, (%rdi,%rax)
    sub    $4, %rax
    jz     loop
```



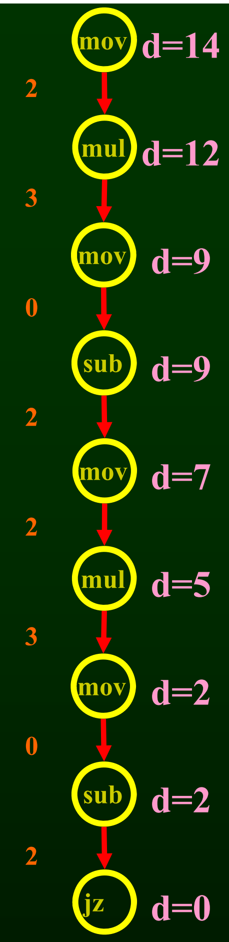
Loop Unrolling

- Rename registers
  - Use different registers in different iterations

- Eliminate unnecessary dependencies
  again, use more registers to eliminate true, anti and output dependencies
  - eliminate dependent-chains of calculations when possible

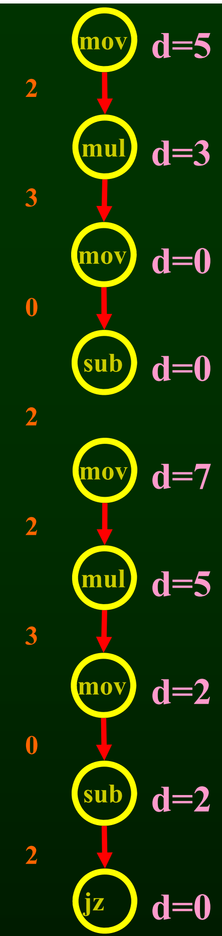
Loop Example

```
loop:
    mov    (%rdi,%rax), %r10
    imul   %r11, %r10
    mov    %r10, (%rdi,%rax)
    sub    $4, %rax
    mov    (%rdi,%rax), %rcx
    imul   %r11, %rcx
    mov    %rcx, (%rdi,%rax)
    sub    $4, %rax
    jz     loop
```



Loop Example

```
loop:
    mov    (%rdi,%rax), %r10
    imul   %r11, %r10
    mov    %r10, (%rdi,%rax)
    sub    $8, %rax
    mov     (%rdi,%rbx), %rcx
    imul   %r11, %rcx
    mov    %rcx, (%rdi,%rbx)
    sub    $8, %rbx
    jz     loop
```

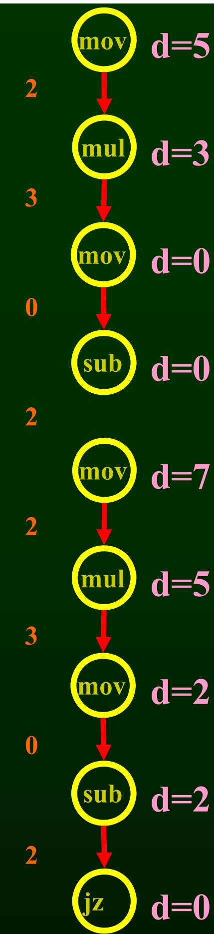


Loop Example

```
loop:
     mov     (%rdi,%rax), %r10
     imul    %r11, %r10
     mov     %r10, (%rdi,%rax)
     sub     $8, %rax
     mov      (%rdi,%rbx), %rcx
     imul    %r11, %rcx
     mov     %rcx, (%rdi,%rbx)
     sub     $8, %rbx
     jz      loop
```

- Schedule (4.5 cycles per iteration

mov		mov			mov		mov			
	mov		mov			mov		mov		
		imul		imul			jz			
			imul		imul			jz		
				imul		imul				
					sub		sub			
						sub		sub		



Outline

- Scheduling for loops
- Loop unrolling
- **Software pipelining**
- Interaction with register allocation
- Hardware vs. Compiler
- loop invariant code motion
- Induction Variable Recognition

Software Pipelining

- Try to overlap multiple iterations so that the slots will be filled
- Find the steady-state window so that:
  - all the instructions of the loop body is executed
  - but from different iterations

Loop Example

- Assembly Code

```
loop:
  mov     (%rdi,%rax), %r10
  imul    %r11, %r10
  mov     %r10, (%rdi,%rax)
  sub     $4, %rax
  jz      loop
```

- Schedule

mov					mov							
	mov					mov						
		mul					jz					
			mul					jz				
				mul								
					sub							
						sub						

Loop Example

- Assembly Code

```
loop:
   mov     (%rdi,%rax), %r10
   imul    %r11, %r10
   mov     %r10, (%rdi,%rax)
   sub     $4, %rax
   jz      loop
```

- Schedule

mov		mov1		mov2	mov	mov3	mov1	mov4	mov2	mov5	mov3	mov6
	mov		mov1		mov2	mov	mov3	mov1	mov4	mov2	ld5	mov3
		mul		mul1		mul2	jz	mul3	jz1	mul4	jz2	mul5
			mul		mul1		mul2	jz	mul3	jz1	mul4	jz2
				mul		mul1		mul2		mul3		mul4
					sub		sub1		sub2		sub3	
						sub		sub1		sub2		sub3

Loop Example

- Assembly Code

```
loop:
  mov    (%rdi,%rax), %r10
  imul   %r11, %r10
  mov    %r10, (%rdi,%rax)
  sub    $4, %rax
  jz     loop
```

- Schedule (2 cycles per iteration)

mov4	mov2
mov1	mov4
mul3	jz1
jz	mul3
mul2	
	sub2
sub1	

Loop Example

- 4 iterations are overlapped
  - value of **`%r11`** don't change
  - 4 regs for **`(%rdi,%rax)`**
  - each addr. incremented by 4*4
  - 4 regs to keep value **`%r10`**
  - Same registers can be reused after 4 of these blocks generate code for 4 blocks, otherwise need to move

mov4	mov2
mov1	mov4
mul3	jz1
jz	mul3
mul2	
	sub2
sub1	

```
loop:
    mov     (%rdi,%rax), %r10
    imul    %r11, %r10
    mov     %r10, (%rdi,%rax)
    sub     $4, %rax
    jz      loop
```

Software Pipelining

- Optimal use of resources
- Need a lot of registers
  - Values in multiple iterations need to be kept
- Issues in dependencies
  - Executing a store instruction in an iteration before branch instruction is executed for a previous iteration (writing when it should not have)
  - Loads and stores are issued out-of-order (need to figure-out dependencies before doing this)
- Code generation issues
  - Generate pre-amble and post-amble code
  - Multiple blocks so no register copy is needed

Outline

- Scheduling for loops
- Loop unrolling
- Software pipelining
- **Interaction with register allocation**
- Hardware vs. Compiler
- Induction Variable Recognition
- loop invariant code motion

Register Allocation and Instruction Scheduling

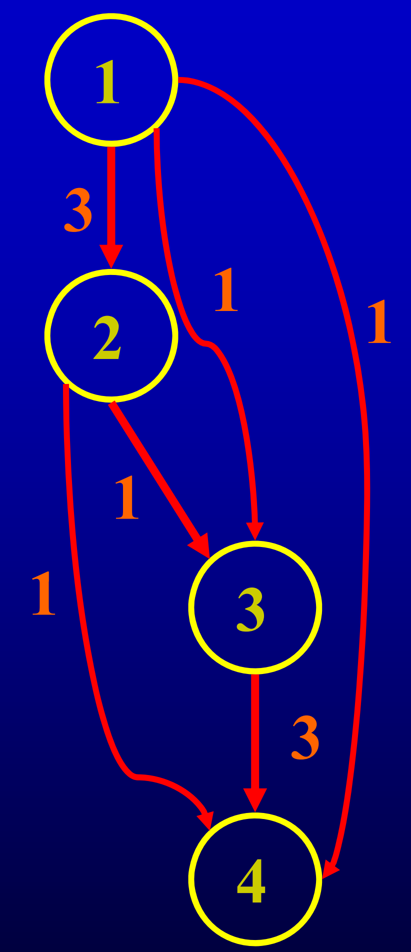
- If register allocation is before instruction scheduling
  - restricts the choices for scheduling

Example

```
1: mov    4(%rbp), %rax
2: add    %rax, %rbx
3: mov    8(%rbp), %rax
4: add    %rax, %rcx
```

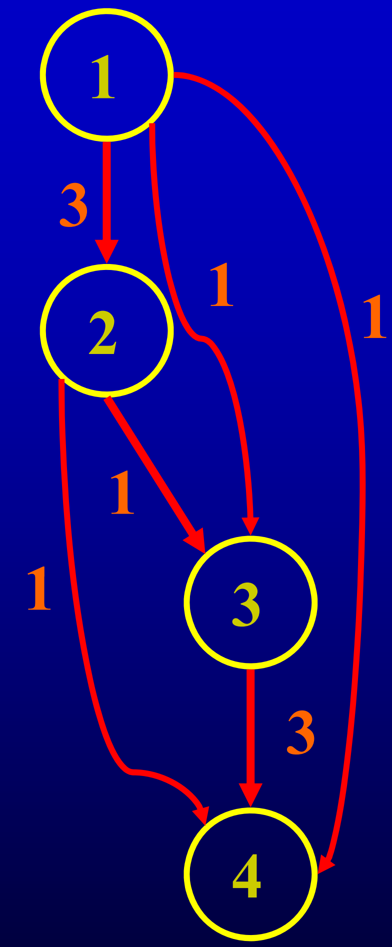
Example

```
1: mov     4(%rbp), %rax
2: add     %rax, %rbx
3: mov     8(%rbp), %rax
4: add     %rax, %rcx
```



Example

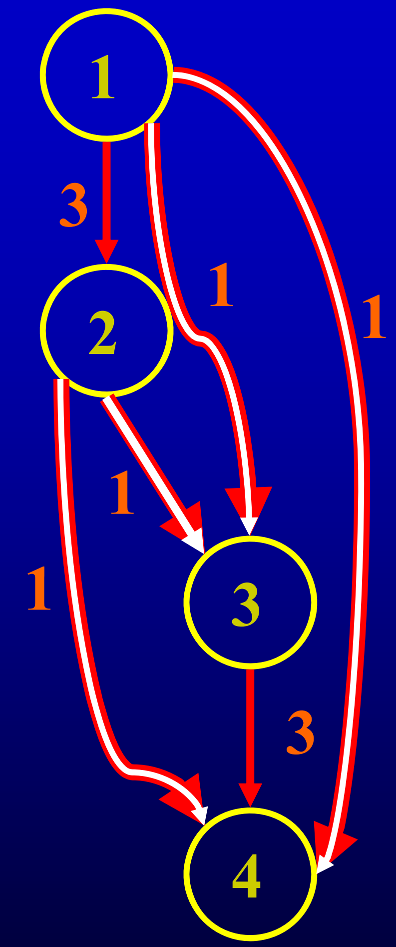
```
1: mov    4(%rbp), %rax
2: add    %rax, %rbx
3: mov    8(%rbp), %rax
4: add    %rax, %rcx
```



ALUop			2			4
MEM 1	1			3		
MEM 2		1			3	

Example

```
1: mov    4(%rbp), %rax
2: add    %rax, %rbx
3: mov    8(%rbp), %rax
4: add    %rax, %rcx
```

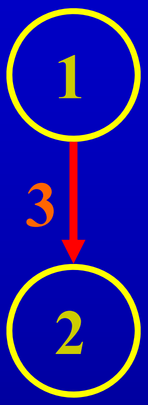


Anti-dependence

How about a different register?

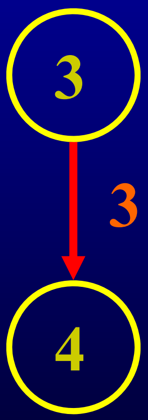
Example

```
1: mov     4(%rbp), %rax
2: add     %rax, %rbx
3: mov     8(%rbp), %r10
4: add     %r10, %rcx
```



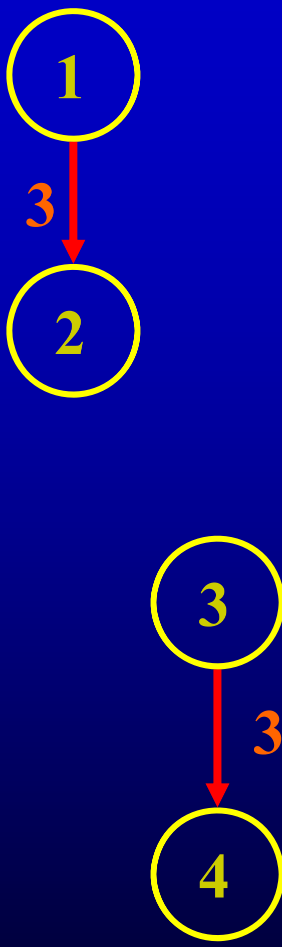
Anti-dependence

How about a different register?



Example

```
1: mov    4(%rbp), %rax
2: add    %rax, %rbx
3: mov    8(%rbp), %r10
4: add    %r10, %rcx
```



ALUop			**2**	**4**
MEM 1	**1**	**3**		
MEM 2		**1**	**3**	

Register Allocation and Instruction Scheduling

- If register allocation is before instruction scheduling
  - restricts the choices for scheduling

Register Allocation and Instruction Scheduling

- If register allocation is before instruction scheduling
  - restricts the choices for scheduling

- If instruction scheduling before register allocation
  - Register allocation may spill registers
  - Will change the carefully done schedule!!!

Outline

- Scheduling for loops
- Loop unrolling
- Software pipelining
- Interaction with register allocation
- **Hardware vs. Compiler**
- Induction Variable Recognition
- loop invariant code motion

Superscalar: Where have all the transistors gone?

- Out of order execution
  - If an instruction stalls, go beyond that and start executing non-dependent instructions
  - Pros:
    - Hardware scheduling
    - Tolerates unpredictable latencies
  - Cons:
    - Instruction window is small

Superscalar: Where have all the transistors gone?

- Register renaming
  - If there is an anti or output dependency of a register that stalls the pipeline, use a different hardware register
  - Pros:
    - Avoids anti and output dependencies
  - Cons:
    - Cannot do more complex transformations to eliminate dependencies

Hardware vs. Compiler

- In a superscalar, hardware and compiler scheduling can work hand-in-hand
- Hardware can reduce the burden when not predictable by the compiler
- Compiler can still greatly enhance the performance
  - Large instruction window for scheduling
  - Many program transformations that increase parallelism
- Compiler is even more critical when no hardware support

  VLIW machines (Itanium, DSPs)

Outline

- Scheduling for loops
- Loop unrolling
- Software pipelining
- Interaction with register allocation
- Hardware vs. Compiler
- **Induction Variable Recognition**
- loop invariant code motion

Induction Variables

- Example

```
i = 200
for j = 1 to 100
  a(i) = 0
  i = i - 1
```

Induction Variables

- Example

```
i = 200
for j = 1 to 100
  a(i) = 0
  i = i - 1
```

Basic Induction variable:

J = 1, 2, 3, 4, …..

Index Variable i in a(i):

I = 200, 199, 198, 197….

Induction Variables

- Example

```
i = 200
for j − 1 to 100
  a(i) = 0
   i = i - 1
```

Basic Induction variable:

J = 1, 2, 3, 4, …..

Index Variable i in a(i):

I = 200, 199, 198, 197…. = 201 - J

Induction Variables

- Example

```
i = 200
for j − 1 to 100
  a(201 - j) = 0
   i = i - 1
```

Basic Induction variable:

J = 1, 2, 3, 4, …..

Index Variable i in a(i):

I = 200, 199, 198, 197…. = 201 - J

Induction Variables

- Example

```
for j − 1 to 100
  a(201 - j) = 0
```

Basic Induction variable:

J = 1, 2, 3, 4, …..

Index Variable i in a(i):

I = 200, 199, 198, 197…. = 201 - J

What are induction variables?

- x is an induction variable of a loop L if
  - variable changes its value every iteration of the loop
  - the value is a function of number of iterations of the loop
- In compilers this function is normally a linear function
  - Example: for loop index variable j, function c*j + d

What can we do with induction variables?

- Use them to perform strength reduction

- Get rid of them

Classification of induction variables

- Basic induction variables
  - Explicitly modified by the same constant amount once during each iteration of the loop
  - Example: loop index variable

- Dependent induction variables
  - Can be expressed in the form: a*x + b where a and be are loop invariant and x is an induction variable
  - Example: **`202 - 2*j`**

Classification of induction variables

- Class of induction variables: All induction variables with same basic variable in their linear equations
- Basis of a class: the basic variable that determines that class

Finding Basic Induction Variables

- Look inside loop nodes
- Find variables whose only modification is of the form **`j = j + d`** where d is a loop constant

Finding Dependent Induction Variables

- Find all the basic induction variables
- Search variable k with a single assignment in the loop
- Variable assignments of the form `k = e op j` or `k = -j` where j is an induction variable and e is loop invariant

Finding Dependent Induction Variables

- Example

```
for i = 1 to 100
    j = i*c
    k = j+1
```

A special case

```
t = 202
for j = 1 to 100
  t = t - 2
  a(j) = t
  t = t - 2
  b(j) = t
```

A special case

```
t = 202
for j = 1 to 100
  t = t - 2
  a(j) = t
  t = t - 2
  b(j) = t
```

⇒

```
u1 = 200
u2 = 202
for j = 1 to 100
  u1 = u1 - 4
  a(j) = u1
  u2 = u2 - 4
  b(j) = u2
```

Outline

- Scheduling for loops
- Loop unrolling
- Software pipelining
- Interaction with register allocation
- Hardware vs. Compiler
- Induction Variable Recognition
- **Loop invariant code motion**

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
for i = 1 to N
  x = x + 1
  for j = 1 to N
    a(i,j) = 100*N + 10*i + j + x
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
for i = 1 to N
  x = x + 1
  for j = 1 to N
    a(i,j) = 100*N + 10*i + j + x
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  for j = 1 to N
     a(i,j) = 100*N + 10*i + j + x
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  for j = 1 to N
     a(i,j) = t1 + 10*i + j + x
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  for j = 1 to N
    a(i,j) = t1 + 10*i + j + x
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  for j = 1 to N
    a(i,j) = t1 + 10*i + j + x
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  t2 = t1 + 10*i + x
  for j = 1 to N
     a(i,j) = t1 + 10*i + j + x
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  t2 = t1 + 10*i + x
  for j = 1 to N
    a(i,j) = t2 + j
```

6.035 Computer Language Engineering
Spring 2010