6.035

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

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
```
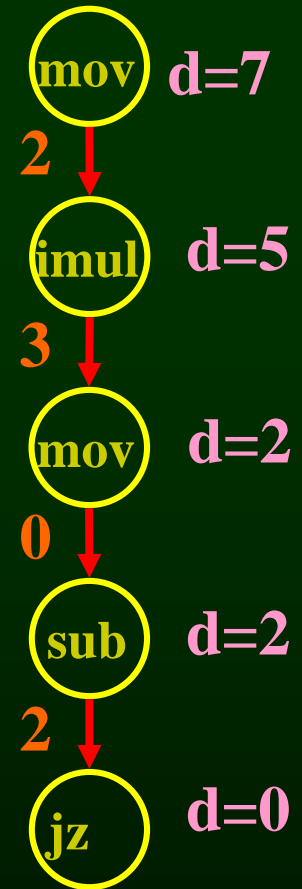
base

offset

Loop Example

- Assembly Code

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

mov   d=7

2

imul   d=5

3

mov   d=2

0

sub   d=2

2

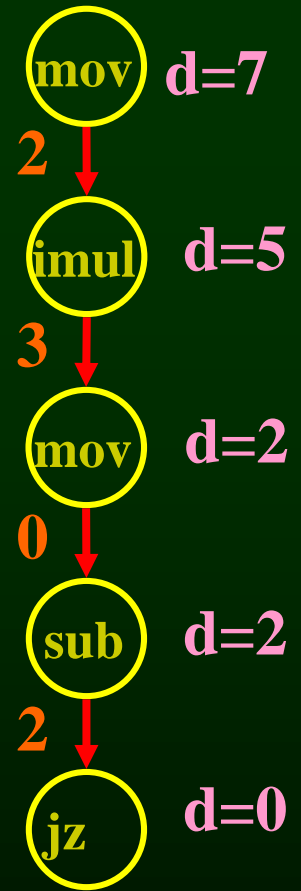jz   d=0

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   d=7

2

imul   d=5

3

mov   d=2

0

sub   d=2

2

jz   d=0

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
```
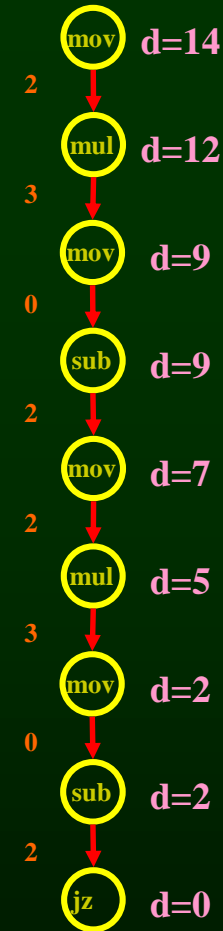
mov d=14

2

mul d=12

3

mov d=9

0

sub d=9

2

mov d=7

2

mul d=5

3

mov d=2

0

sub d=2

2

jz d=0

- Schedule (8 cycles per iteration)

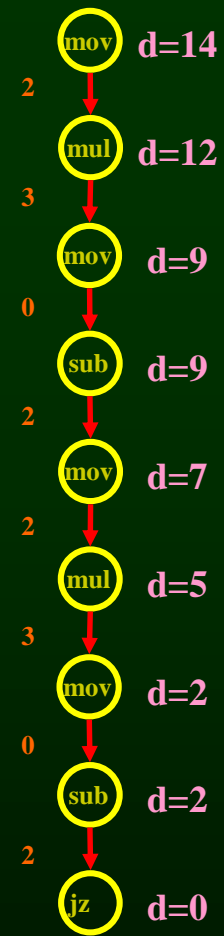| mov					mov	mov					mov			
	mov				mov	mov					mov			
		imul							imul				bge	
			imul							imul				bge
				imul							imul			
				sub							sub			
					sub							sub		

Loop Unrolling

- Rename registers
  - Use different registers in different iterations

Loop Example

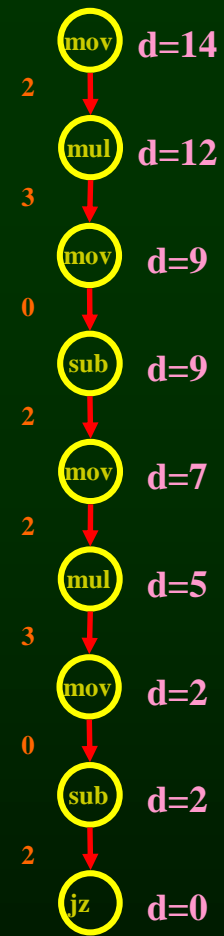```
loop:
    mov   (%rdi,%rax), %r10
    imul   %r11, %r10
    mov   %r10, (%rdi,%rax)
    sub $4, %rax
    mov   (%rdi,%rax), %r10
    imul   %r11, %r10
    mov   %r10, (%rdi,%rax)
    sub $4, %rax
    jz   loop
```

mov — d=14

2

mul — d=12

3

mov — d=9

0

sub — d=9

2

mov — d=7

2

mul — d=5

3

mov — d=2

0

sub — d=2

2

jz — d=0

# 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
```

mov d=14

2

mul d=12

3

mov d=9

0

sub d=9

2

mov d=7

2

mul d=5

3

mov d=2

0

sub d=2

2

jz d=0

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
```
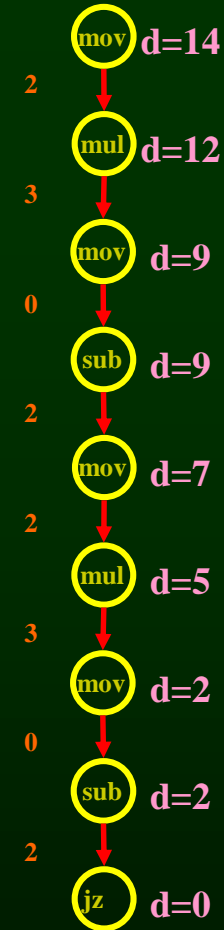
mov d=14

2

mul d=12

3

mov d=9

0

sub d=9

2

mov d=7

2

mul d=5

3

mov d=2

0

sub d=2

2

jz d=0

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
```
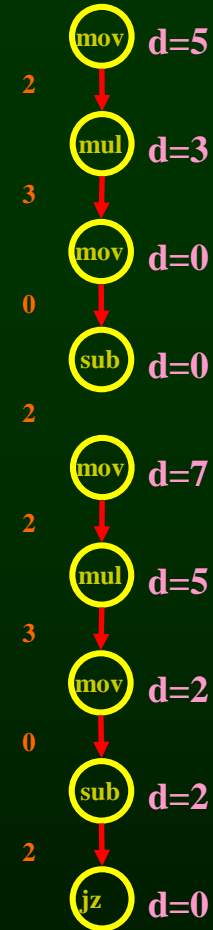
mov d=5

2

mul d=3

3

mov d=0

0

sub d=0

2

mov d=7

2

mul d=5

3

mov d=2

0

sub d=2

2

jz d=0

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
```
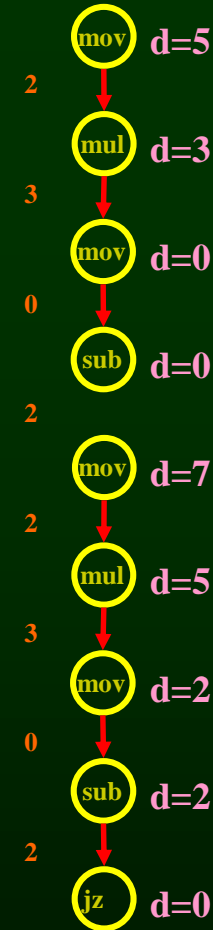
- Schedule (4.5 cycles per iteration

mov d=5
2
mul d=3
3
mov d=0
0
sub d=0
2
mov d=7
2
mul d=5
3
mov d=2
0
sub d=2
2
jz   d=0

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			mov		mov1				
	mov		mov1			mov		mov1			
		mul		mul1			jz		jz1		
			mul		mul1			jz		jz1	
				mul		mul1					
					sub		sub1				
						sub		sub1			

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		mov1		mov2			
	mov		mov1		mov2	mov		mov1		mov2		
		mul		mul1		mul2	jz		jz1		jz2	
			mul		mul1		mul2	jz		jz1		jz2
				mul		mul1		mul2				
					sub		sub1		sub2			
						sub		sub1		sub2		

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		mov2		mov3
	mov		mov1		mov2	mov	mov3	mov1		mov2		mov3
	mul		mul1		mul2	jz	mul3	jz1		jz2	
		mul		mul1		mul2	jz	mul3	jz1		jz2
		mul		mul1		mul2		mul3		
			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

mov		mov1		mov2	mov	mov3	mov1	mov4	mov2		mov3	
	mov		mov1		mov2	mov	mov3	mov1	mov4	mov2		mov3
		mul		mul1		mul2	jz	mul3	jz1	mul4	jz2	
			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

mov		mov1		mov2	mov	mov3	mov1	mov4	mov2	mov5	mov3	
	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

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

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
```
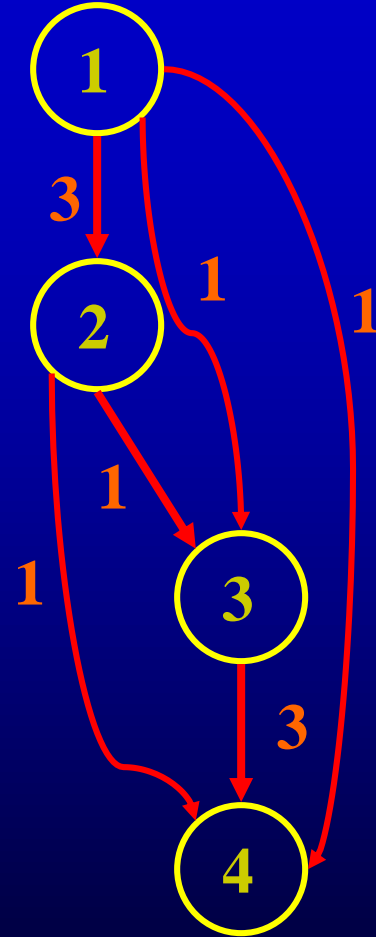
Example

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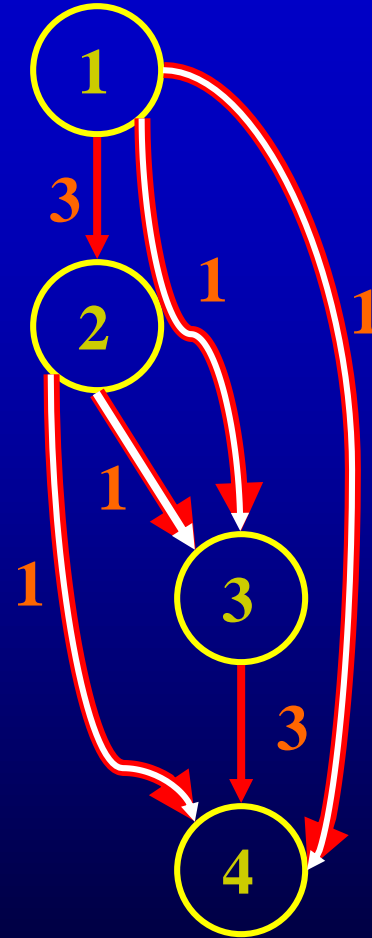
Example

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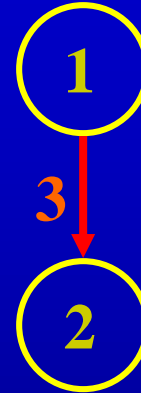

ALUop			2			4
MEM 1	1			3		
MEM 2		1			3	

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
```
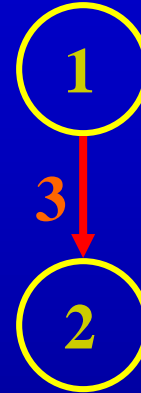
Anti-dependence

How about a different register?

Example

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

	1
	3
	2

	3
	3
	4

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

Out Of Order Hardware Scheduling

- Out of order execution
  - If an instruction stalls, go beyond that and start executing non-dependent instructions
  - Pros:
    - Tolerates unpredictable latencies
    - More information available at runtime than compile time
  - Cons:
    - Exploit parallelism only within instruction window

Register Renaming

- 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