6.035

Fall 2005

Lectures 14 & 15: Instruction Scheduling

Simple Machine Model

- Instructions are executed in sequence
  - Fetch, decode, execute, store results
  - One instruction at a time
- For branch instructions, start fetching from a different location if needed
  - Check branch condition
  - Next instruction may come from a new location given by the branch instruction

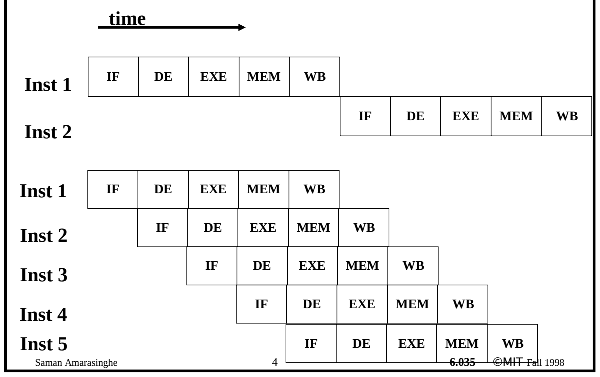
Simple Execution Model

- 5 Stage pipe-line

fetch	**decode**	**execute**	**memory**	**writeback**

- Fetch: get the next instruction
- Decode: figure-out what that instruction is
- Execute: Perform ALU operation
  - address calculation in a memory op
- Memory: Do the memory access in a mem. Op.
- Write Back: write the results back

Simple Execution Model



From a Simple Machine Model to a Real Machine Model

- Many pipeline stages

– Pentium	5
– Pentium Pro	10
– Pentium IV (130nm)	20
– Pentium IV (90nm)	31

- Different instructions taking different amount of time to execute
- Hardware to stall the pipeline if an instruction uses a result that is not ready

Real Machine Model cont.

- Most modern processors have multiple execution units (superscalar)
  - If the instruction sequence is correct, multiple operations will happen in the same cycles
  - Even more important to have the right instruction sequence

Constraints On Scheduling

- Data dependencies
- Control dependencies
- Resource Constraints

Data Dependency between Instructions

- If two instructions access the same variable, they can be dependent
- Kind of dependencies
  - True: write → read
  - Anti: read → write
  - Output: write → write
- What to do if two instructions are dependent.
  - The order of execution cannot be reversed
  - Reduce the possibilities for scheduling

Computing Dependencies

- For basic blocks, compute dependencies by walking through the instructions
- Identifying register dependencies is simple
  - is it the same register?
- For memory accesses
  - simple: base + offset1 ?= base + offset2
  - data dependence analysis: a[2i] ?= a[2i+1]
  - interprocedural analysis: global ?= parameter
  - pointer alias analysis: p1 ?= p

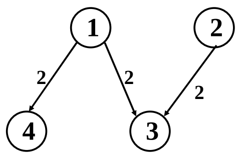
Representing Dependencies

- Using a dependence DAG, one per basic block
- Nodes are instructions, edges represent dependencies

Representing Dependencies

- Using a dependence DAG, one per basic block
- Nodes are instructions, edges represent dependencies

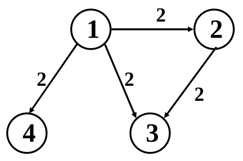
```
1: r2 = *(r1 + 4)
2: r3 = *(r1 + 8)
3: r4 = r2 + r3
4: r5 = r2 - 1
```



- Edge is labeled with Latency:
  - v(i → j) = delay required between initiation times of i and j minus the execution time required by i

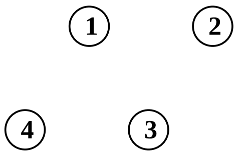
Example

```
1: r2 = *(r1 + 4)
2: r3 = *(r2 + 4)
3: r4 = r2 + r3
4: r5 = r2 - 1
```



Another Example

```
1: r2 = *(r1 + 4)
2: *(r1 + 4) = r3
3: r3 = r2 + r3
4: r5 = r2 - 1
```



Control Dependencies and Resource Constraints

- For now, lets only worry about basic blocks
- For now, lets look at simple pipelines

Example

	Results In
`1: lea  var_a, %rax`	1 cycle
`2: add  $4, %rax`	1 cycle
`3: inc  %r11`	1 cycle
`4: mov  4(%rsp), %r10`	3 cycles
`5: add  %r10, 8(%rsp)`	
`6: and  16(%rsp), %rbx`	4 cycles
`7: imul %rax, %rbx`	3 cycles

1	2	3	4	st	st	5	6	st	st	st	7

List Scheduling Algorithm

- Idea
  - Do a topological sort of the dependence DAG
  - Consider when an instruction can be scheduled without causing a stall
  - Schedule the instruction if it causes no stall and all its predecessors are already scheduled
- Optimal list scheduling is NP-complete
  - Use heuristics when necessary

List Scheduling Algorithm

- Create a dependence DAG of a basic block
- Topological Sort

```
READY = nodes with no predecessors
Loop until READY is empty
    Schedule each node in READY when no stalling
    Update READY
```

Heuristics for selection

- Heuristics for selecting from the READY list
  - pick the node with the longest path to a leaf in the dependence graph
  - pick a node with most immediate successors
  - pick a node that can go to a less busy pipeline (in a superscalar)

Heuristics for selection

- pick the node with the longest path to a leaf in the dependence graph
- Algorithm (for node x)
  - If no successors $d_x = 0$
  - $d_x = \mathrm{MAX}(d_y + c_{xy})$ for all successors y of x
  - reverse breadth-first visitation order

Heuristics for selection

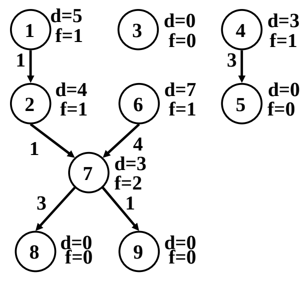
- pick a node with most immediate successors
- Algorithm (for node x):
  - f_x = number of successors of x

Example

```
                                 Results In
1: lea  var_a, %rax              1 cycle
2: add  $4, %rax                 1 cycle
3: inc  %r11                     1 cycle
4: mov  4(%rsp), %r10            3 cycles
5: add  %r10, 8(%rsp)
6: and  16(%rsp), %rbx           4 cycles
7: imul %rax, %rbx               3 cycles
8: mov  %rbx, 16(%rsp)
9: lea  var_b, %rax
```

Example

READY = { }



6	1	2	4	7	3	5	8	9

Example

```
                                 Results In
1: lea  var_a, %rax              1 cycle
2: add  $4, %rax                 1 cycle
3: inc  %r11                     1 cycle
4: mov  4(%rsp), %r10            3 cycles
5: add  %r10, 8(%rsp)
6: and  16(%rsp), %rbx           4 cycles
7: imul %rax, %rbx               3 cycles
8: mov  %rbx, 16(%rsp)
9: lea  var_b, %rax
```

1	2	3	4	st	st	5	6	st	st	st	7	8	9

6	1	2	4	7	3	5	8	9

14 cycles vs 9 cycles

Resource Constraints

- Modern machines have many resource constraints
- Superscalar architectures:
  - can run few parallel operations
  - But have constraints

Resource Constraints of a Superscalar Processor

- Example:
  - One fully pipelined reg-to-reg unit
    - All integer operations taking one cycle

  In parallel with
  - One fully pipelined memory-to/from-reg unit
    - Data loads take two cycles
    - Data stores teke one cycle

List Scheduling Algorithm with resource constraints

- Represent the superscalar architecture as multiple pipelines
  - Each pipeline represent some resource
- Example
  - One single cycle reg-to-reg ALU unit
  - One two-cycle pipelined reg-to/from-memory unit

ALU							
MEM 1							
MEM 2							

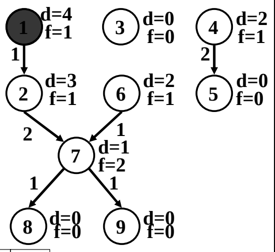
List Scheduling Algorithm with resource constraints

- Create a dependence DAG of a basic block
- Topological Sort

```
READY = nodes with no predecessors
Loop until READY is empty
    Let n ∈READY be the node with the highest priority
    Schedule n in the earliest slot
        that satisfies precedence + resource constraints
    Update READY
```

Example

```
1: lea  var_a, %rax
2: add  4(%rsp), %rax
3: inc  %r11
4: mov  4(%rsp), %r10
5: mov  %r10, 8(%rsp)
6: and  $0x00ff, %rbx
7: imul %rax, %rbx
8: lea  var_b, %rax
9: mov  %rbx, 16(%rsp)
```

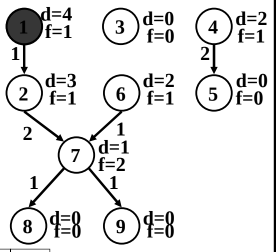


READY = { 1, 6, 4, 3 }

ALUop							
MEM 1							
MEM 2							

Example

```
1: lea  var_a, %rax
2: add  4(%rsp), %rax
3: inc  %r11
4: mov  4(%rsp), %r10
5: mov  %r10, 8(%rsp)
6: and  $0x00ff, %rbx
7: imul %rax, %rbx
8: lea  var_b, %rax
9: mov  %rbx, 16(%rsp)
```

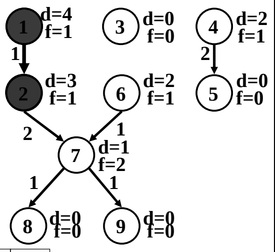


READY = { 1, 6, 4, 3 }

ALUop	1						
MEM 1							
MEM 2							

Example

```
1: lea  var_a, %rax
2: add  4(%rsp), %rax
3: inc  %r11
4: mov  4(%rsp), %r10
5: mov  %r10, 8(%rsp)
6: and  $0x00ff, %rbx
7: imul %rax, %rbx
8: lea  var_b, %rax
9: mov  %rbx, 16(%rsp)
```

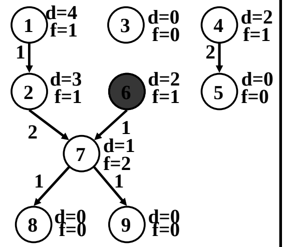


READY = { 2, 6, 4, 3 }

ALUop	1						
MEM 1		2					
MEM 2			2				

Example

```
1: lea  var_a, %rax
2: add  4(%rsp), %rax
3: inc  %r11
4: mov  4(%rsp), %r10
5: mov  %r10, 8(%rsp)
6: and  $0x00ff, %rbx
7: imul %rax, %rbx
8: lea  var_b, %rax
9: mov  %rbx, 16(%rsp)
```

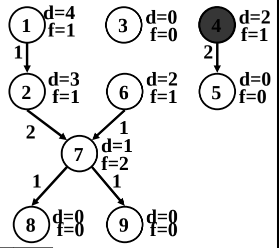


READY = { 6, 4, 3 }

ALUop	1	6					
MEM 1		2					
MEM 2			2				

Example

```
1: lea  var_a, %rax
2: add  4(%rsp), %rax
3: inc  %r11
4: mov  4(%rsp), %r10
5: mov  %r10, 8(%rsp)
6: and  $0x00ff, %rbx
7: imul %rax, %rbx
8: lea  var_b, %rax
9: mov  %rbx, 16(%rsp)
```

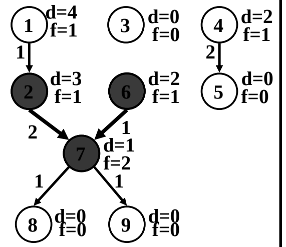


READY = { 4, 7, 3 }

ALUop	1	6					
MEM 1	4	2					
MEM 2		4	2				

Example

```
1: lea  var_a, %rax
2: add  4(%rsp), %rax
3: inc  %r11
4: mov  4(%rsp), %r10
5: mov  %r10, 8(%rsp)
6: and  $0x00ff, %rbx
7: imul %rax, %rbx
8: lea  var_b, %rax
9: mov  %rbx, 16(%rsp)
```

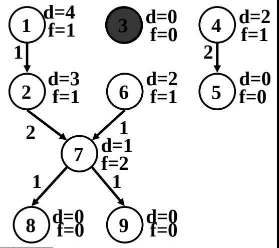


READY = { 7, 3, 5 }

ALUop	1	6		7			
MEM 1	4	2					
MEM 2		4	2				

Example

```
1: lea  var_a, %rax
2: add  4(%rsp), %rax
3: inc  %r11
4: mov  4(%rsp), %r10
5: mov  %r10, 8(%rsp)
6: and  $0x00ff, %rbx
7: imul %rax, %rbx
8: lea  var_b, %rax
9: mov  %rbx, 16(%rsp)
```

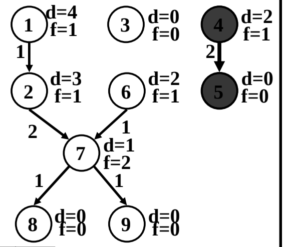


READY = { 3, 5, 8, 9 }

ALUop	1	6	3	7			
MEM 1	4	2					
MEM 2		4	2				

Example

```
1: lea  var_a, %rax
2: add  4(%rsp), %rax
3: inc  %r11
4: mov  4(%rsp), %r10
5: mov  %r10, 8(%rsp)
6: and  $0x00ff, %rbx
7: imul %rax, %rbx
8: lea  var_b, %rax
9: mov  %rbx, 16(%rsp)
```

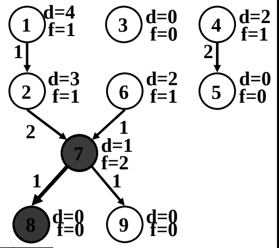


READY = { 5, 8, 9 }

ALUop	1	6	3	7			
MEM 1	4	2	5				
MEM 2		4	2				

Example

```
1: lea  var_a, %rax
2: add  4(%rsp), %rax
3: inc  %r11
4: mov  4(%rsp), %r10
5: mov  %r10, 8(%rsp)
6: and  $0x00ff, %rbx
7: imul %rax, %rbx
8: lea  var_b, %rax
9: mov  %rbx, 16(%rsp)
```

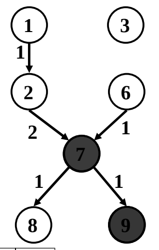


READY = { 8, 9 }

ALUop	1	6	3	7	8		
MEM 1	4	2	5				
MEM 2		4	2				

Example

```
1: lea  var_a, %rax
2: add  4(%rsp), %rax
3: inc  %r11
4: mov  4(%rsp), %r10
5: mov  %r10, 8(%rsp)
6: and  $0x00ff, %rbx
7: imul %rax, %rbx
8: lea  var_b, %rax
9: mov  %rbx, 16(%rsp)
```



READY = { }

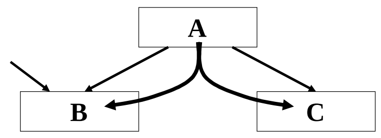
ALUop	1	6	3	7	8		
MEM 1	4	2	5		9		
MEM 2		4	2				

Scheduling across basic blocks

- Number of instructions in a basic block is small
  - Cannot keep a multiple units with long pipelines busy by just scheduling within a basic block
- Need to handle control dependence
  - Scheduling constraints across basic blocks
  - Scheduling policy

Moving across basic blocks

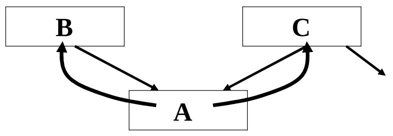
- Downward to adjacent basic block



- A path to B that does not execute A?

Moving across basic blocks

- Upward to adjacent basic block



- A path from C that does not reach A?

Control Dependencies

- Constraints in moving instructions across basic blocks

```
if ( . . . )
    a = b op c
```

Control Dependencies

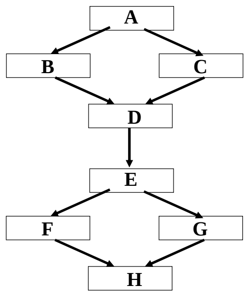
- Constraints in moving instructions across basic blocks

```
If ( valid address? )
    d = *(a1)
```

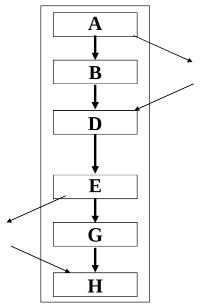
Trace Scheduling

- Find the most common trace of basic blocks
  - Use profile information
- Combine the basic blocks in the trace and schedule them as one block
- Create clean-up code if the execution goes off-trace

Trace Scheduling

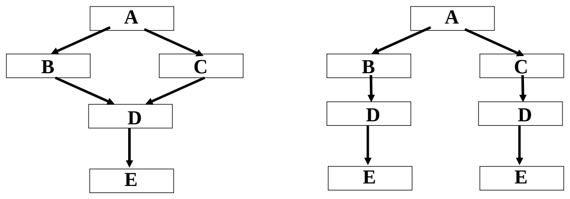


Trace Scheduling

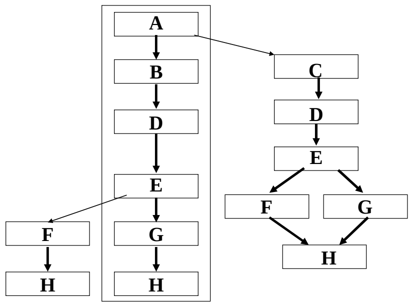


Large Basic Blocks via Code Duplication

- Creating large extended basic blocks by duplication
- Schedule the larger blocks



Trace Scheduling



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

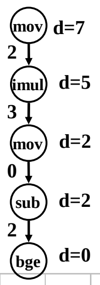
- Source Code

```
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
  bge   loop
```

Loop Example



- Assembly Code

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

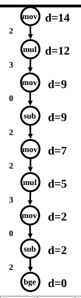
- Schedule (9 cycles per iteration)

mov					mov							
	mov					mov						
		imul					bge					
			imul					bge				
				imul								
					sub							
						sub						

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

- Schedule (8 cycles per iteration)

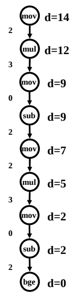
mov					mov		mov					mov			
	mov					mov		mov					mov		
		imul							imul					bge	
			imul							imul					bge
				imul							imul				
					sub							sub			
						sub							sub		

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), %rcx
    imul  %r11, %rcx
    mov   %rcx, (%rdi,%rax)
    sub   $4, %rax
    bge   loop
```

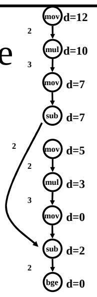


Loop Unrolling

- 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   $8, %rax
    mov    (%rdi,%rbx), %rcx
    imul  %r11, %rcx
    mov   %rcx, (%rdi,%rbx)
    sub   $8, %rbx
    bge   loop
```



- Schedule (4.5 cycles per iteration

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

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
    bge   loop
```

- Schedule

mov		mov1		mov2	st	mov3	st1	mov4	st2	mov5	st3	mov6
	mov		mov1		mov2	mov	mov3	mov1	mov4	mov2	ld5	mov3
		mul		mul1		mul2	bge	mul3	bge1	mul4	bge2	mul5
			mul		mul1		mul2	bge	mul3	bge1	mul4	bge2
				mul		mul1		mul2		mul3		mul4
					sub		sub1		sub2		sub3	
						sub		sub1		sub2		sub3

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

mov3	st1
mov	mov3
mul2	bge
	mul2
mul1	
	sub1
sub	

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

Software Pipelining

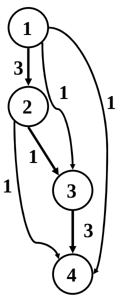
- Optimal use of resources
- Need a lot of registers
  - Values in multiple iterations need to be kept
- Issues in dependencies
  - Executing a store instruction in an iteration before branch instruction is executed for a previous iteration (writing when it should not have)
  - Loads and stores are issued 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

Register Allocation and Instruction Scheduling

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

Example

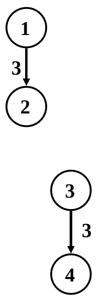
```
1: mov    4(%rbp), %rax
2: add    %rax, %rbx
3: mov    8(%rbp), %rax
4: add    %rax, %rcx
```



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), %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!!!

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)