6.035

Spring 2010

Introduction to Code Optimization

Instruction Scheduling

Outline

- Modern architectures
- Introduction to instruction scheduling
- List scheduling
- Resource constraints
- Scheduling across basic blocks
- Trace scheduling

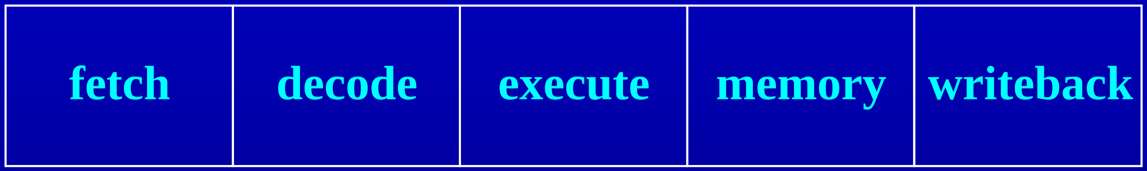
Simple Machine Model

- 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

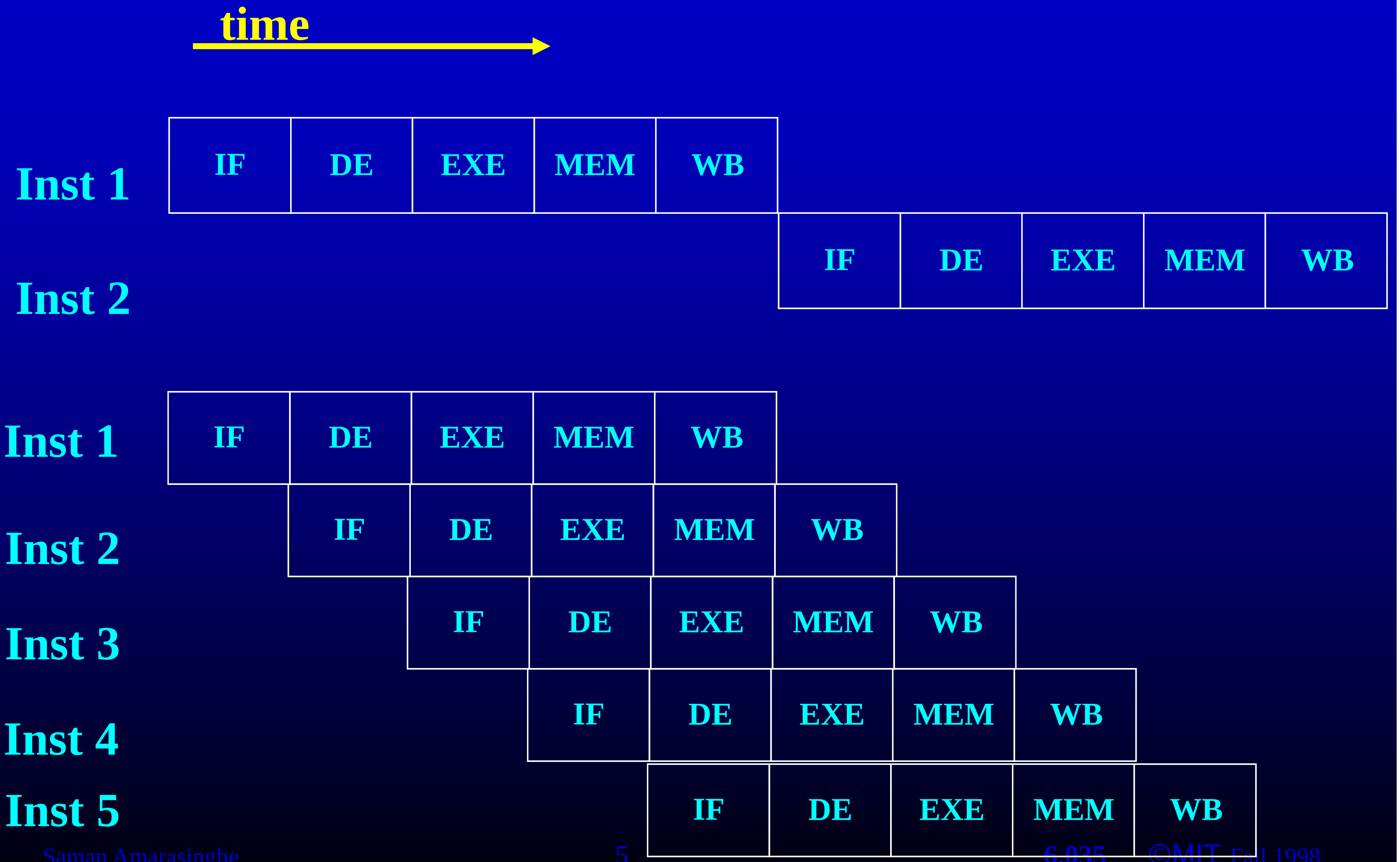
Simple Execution Model

- 5 Stage pipe-line



- Fetch: 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



Outline

- Modern architectures
- Introduction to instruction scheduling
- List scheduling
- Resource constraints
- Scheduling across basic blocks
- Trace scheduling

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
– Core 2 Duo	14

- 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 cores
  - Will deal with multicores next week
- Each core has 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

Instruction Scheduling

- Reorder instructions so that pipeline stalls are minimized

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 $\rightarrow$ read
  - Anti: read $\rightarrow$ write
  - Output: write $\rightarrow$ 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→foo ?= p2→foo

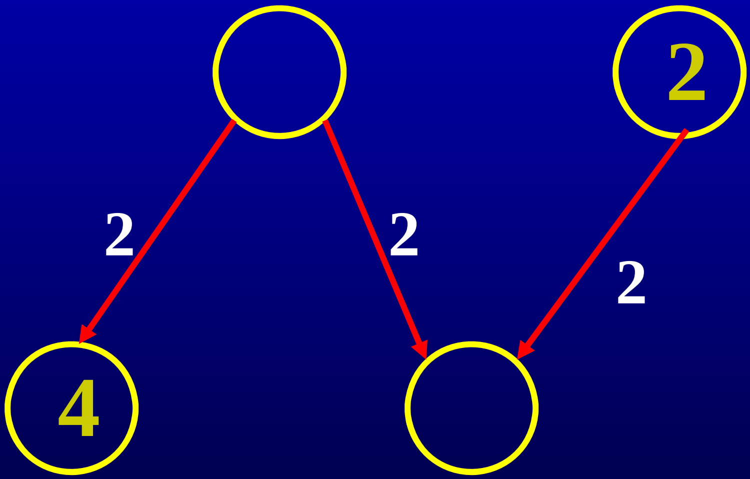
Representing Dependencies

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

Representing Dependencies

- Using$_{\text{a de}}$ pendence DAG, one per basic block
- Nodes are instructions, edges represent dependencies

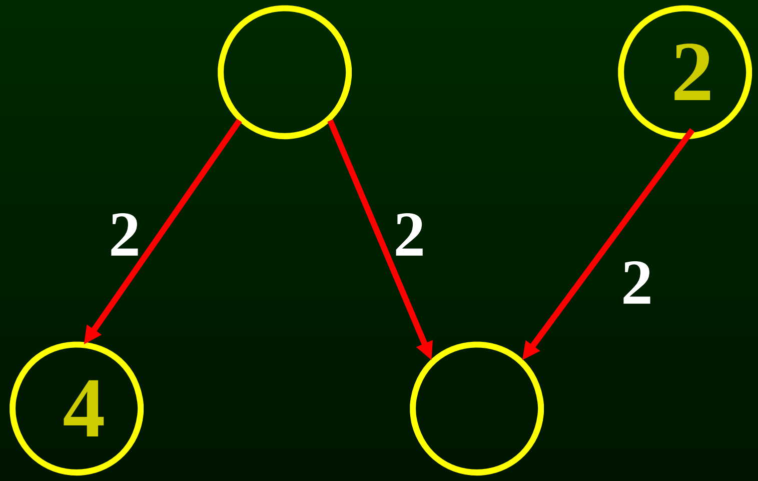
```
1: r2 = *(r1 + 4)
2: r3 = *(r1 + 8)
3: r4 = r2 + r3
4: r5 = r2 - 1
```



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

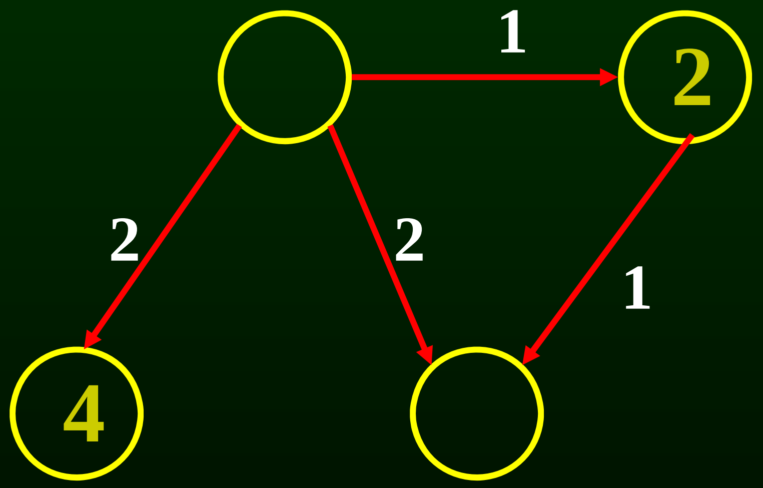
Example

```
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

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

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

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

Outline

- Modern architectures
- Introduction to instruction scheduling
- List scheduling
- Resource constraints
- Scheduling across basic blocks
- Trace scheduling

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

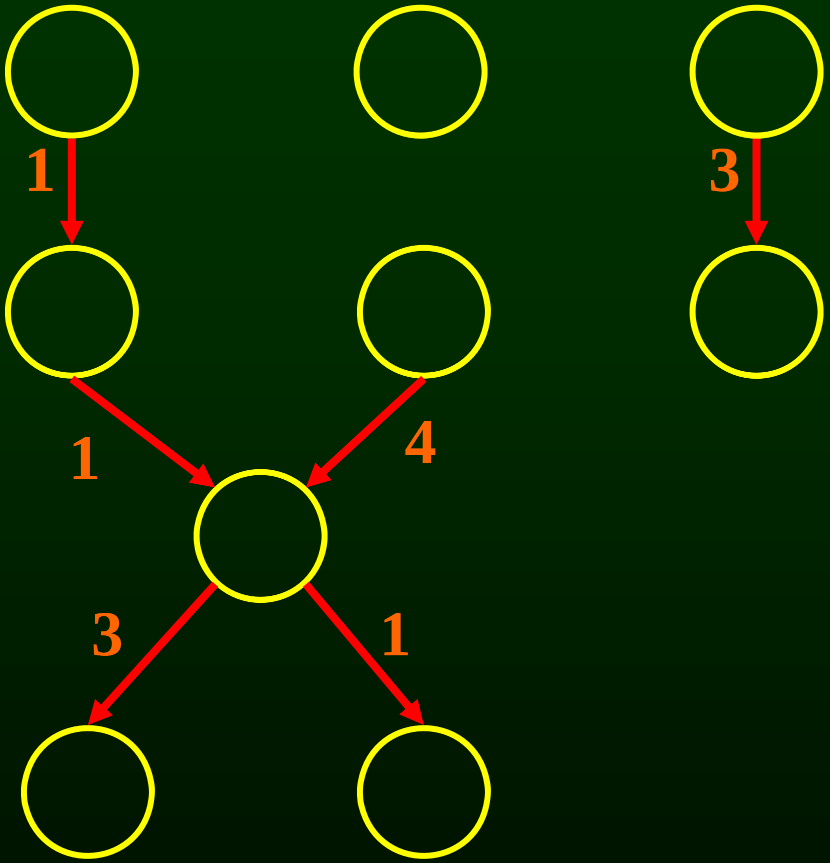
- pick a node with most immediate successors
- Algorithm (for node x):
  - f_x = number of successors of x

Example

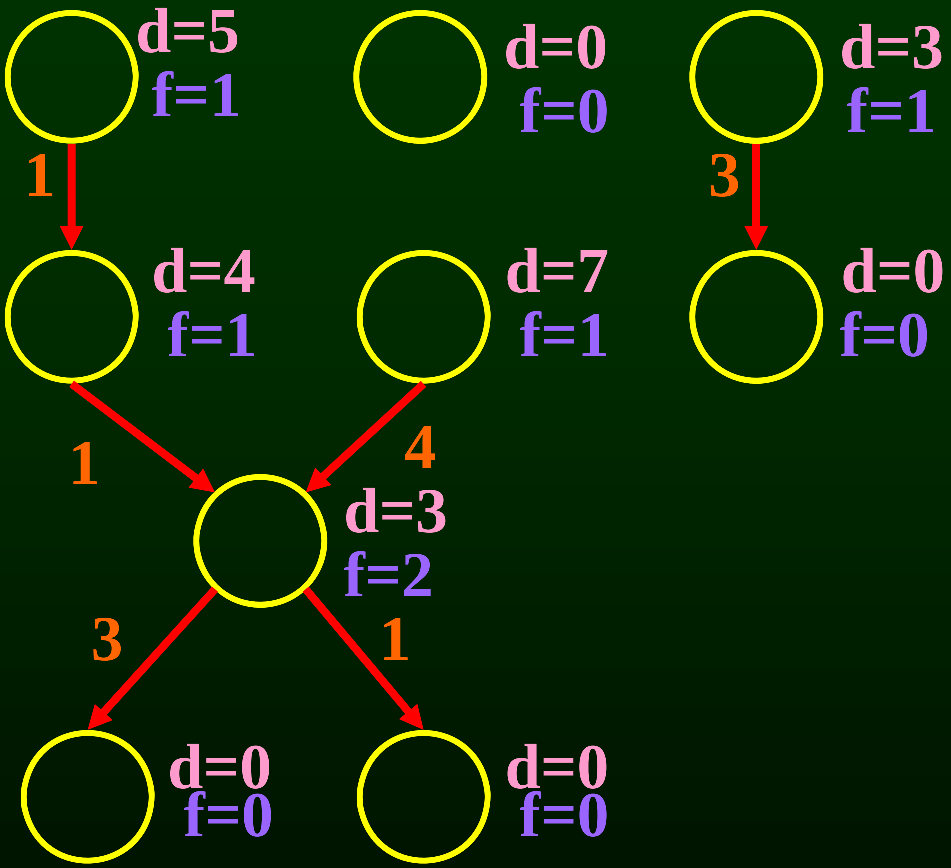
	Results In
`1: lea  var_a, %rax`	1 cycle
`2: add  $4, %rax`	1 cycle
`3: inc  %r11`	1 cycle
`4: mov  4(%rsp), %r10`	3 cycles
`5: add  %r10, 8(%rsp)`	
`6: and  16(%rsp), %rbx`	4 cycles
`7: imul %rax, %rbx`	3 cycles
`8: mov  %rbx, 16(%rsp)`	
`9: lea  var_b, %rax`	

Example

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

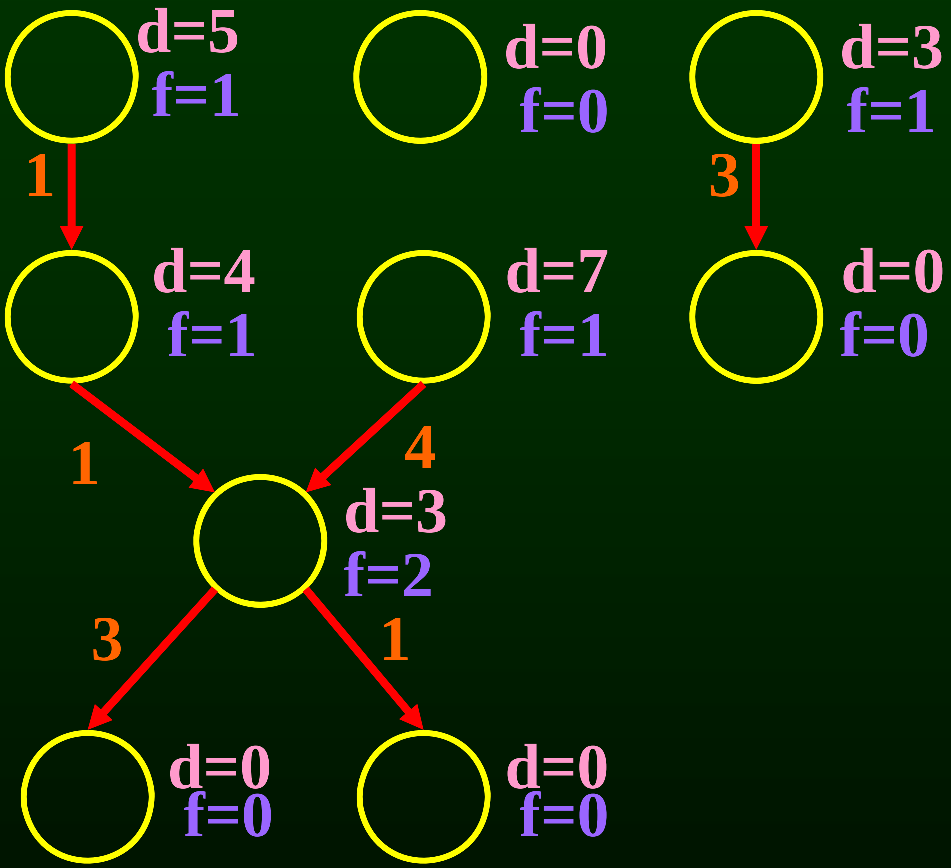


Example



Example

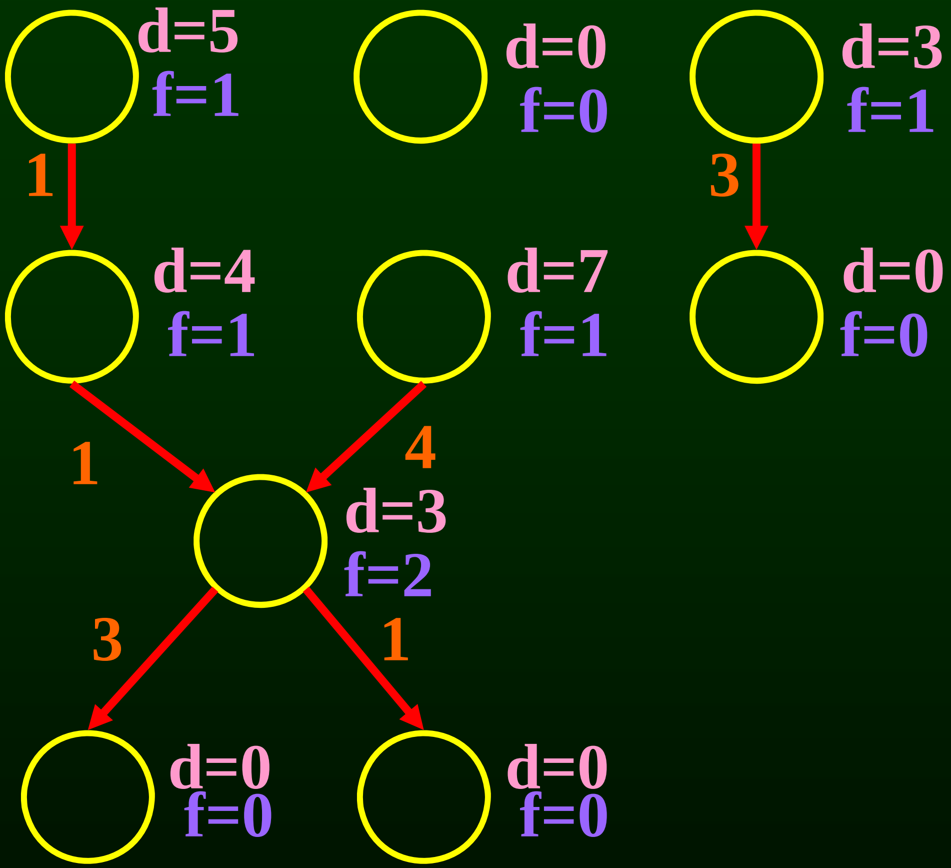
READY = { }



Example

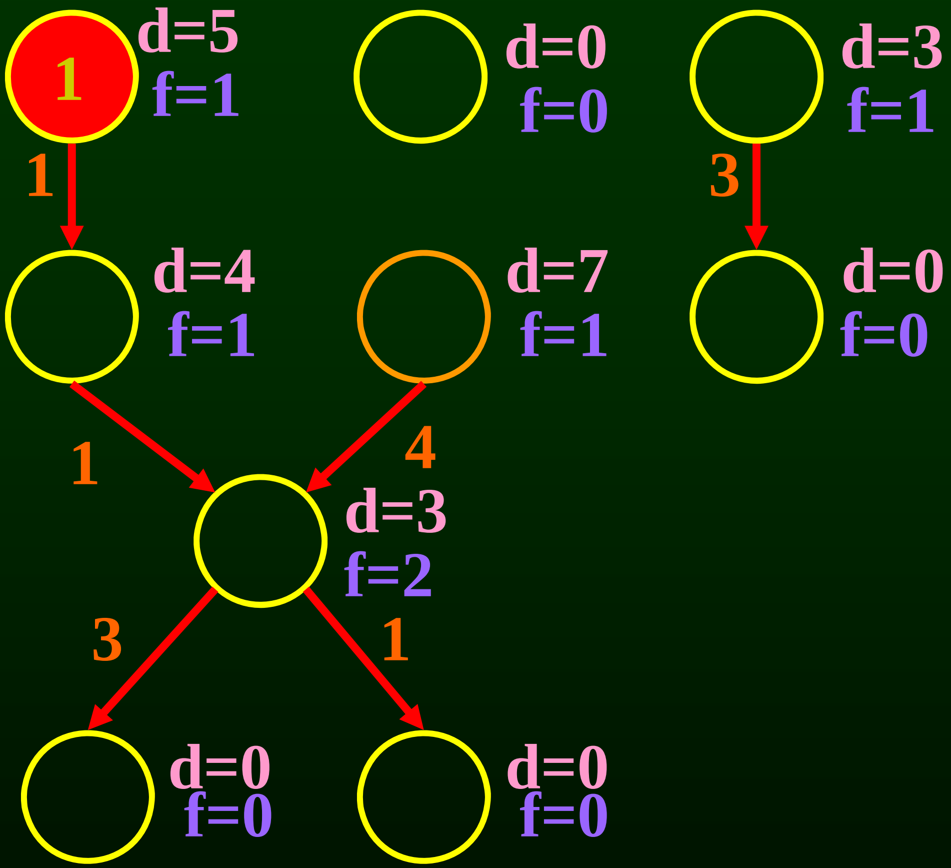
1, 3, 4, 6

READY = { }



Example

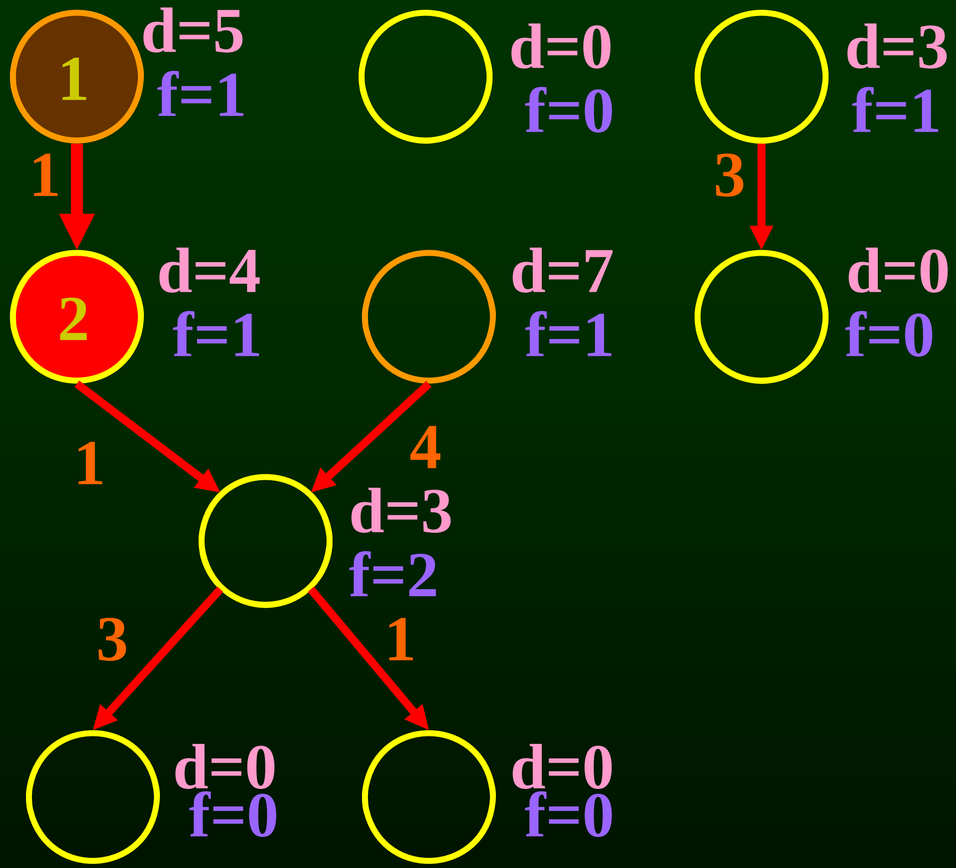
READY = { 1, 4, 3 }

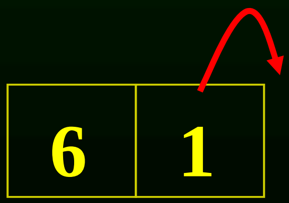


$\boxed{6}$

Example

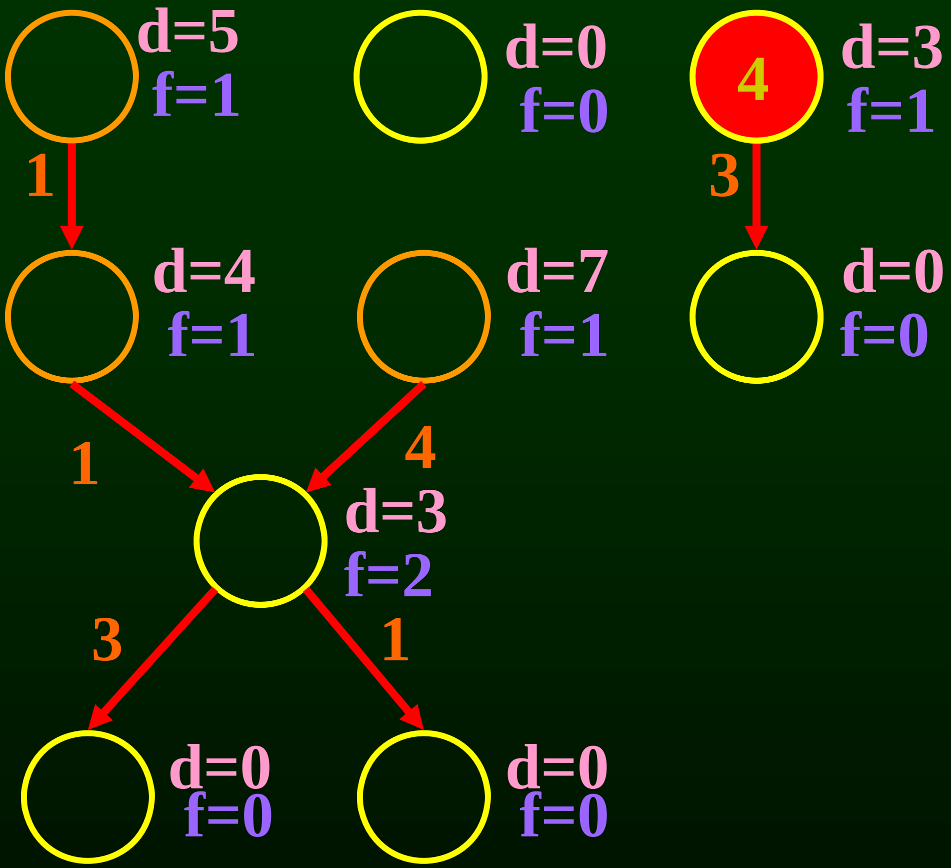
READY = { 2, 4, 3 }





Example

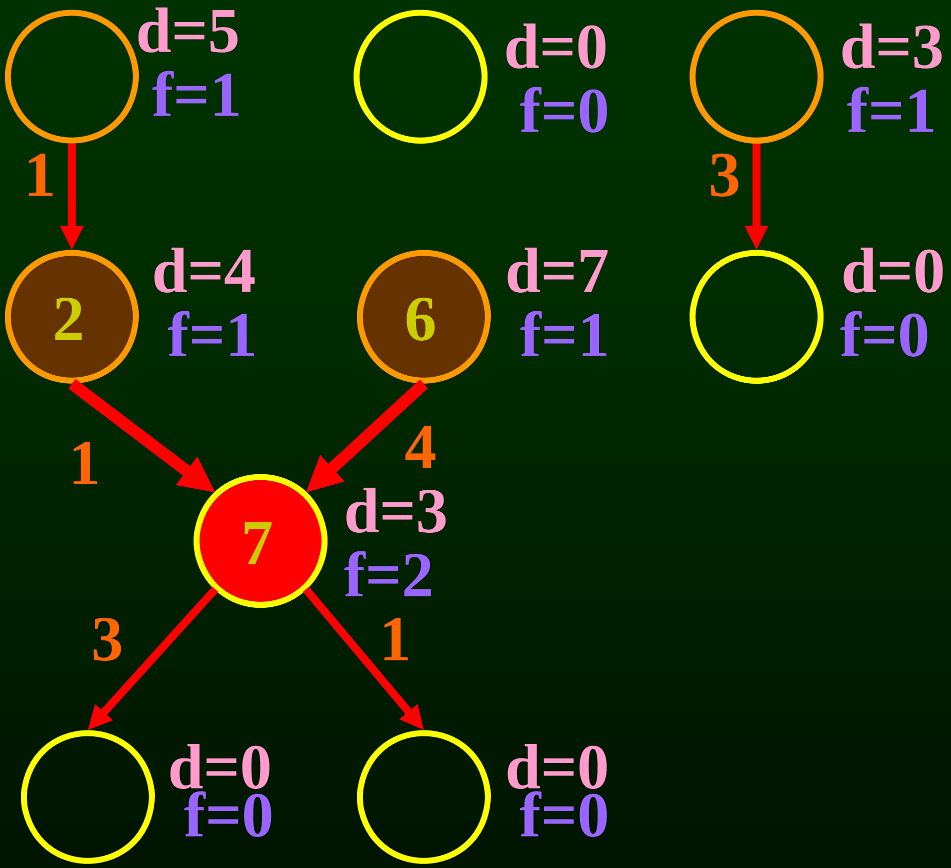
READY = { 7, **4**, 3 }

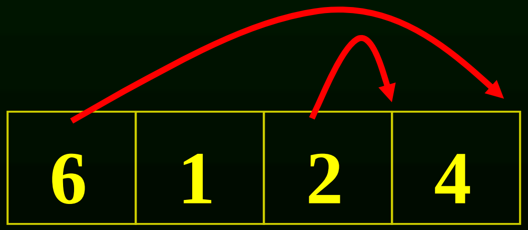


6	1	2

Example

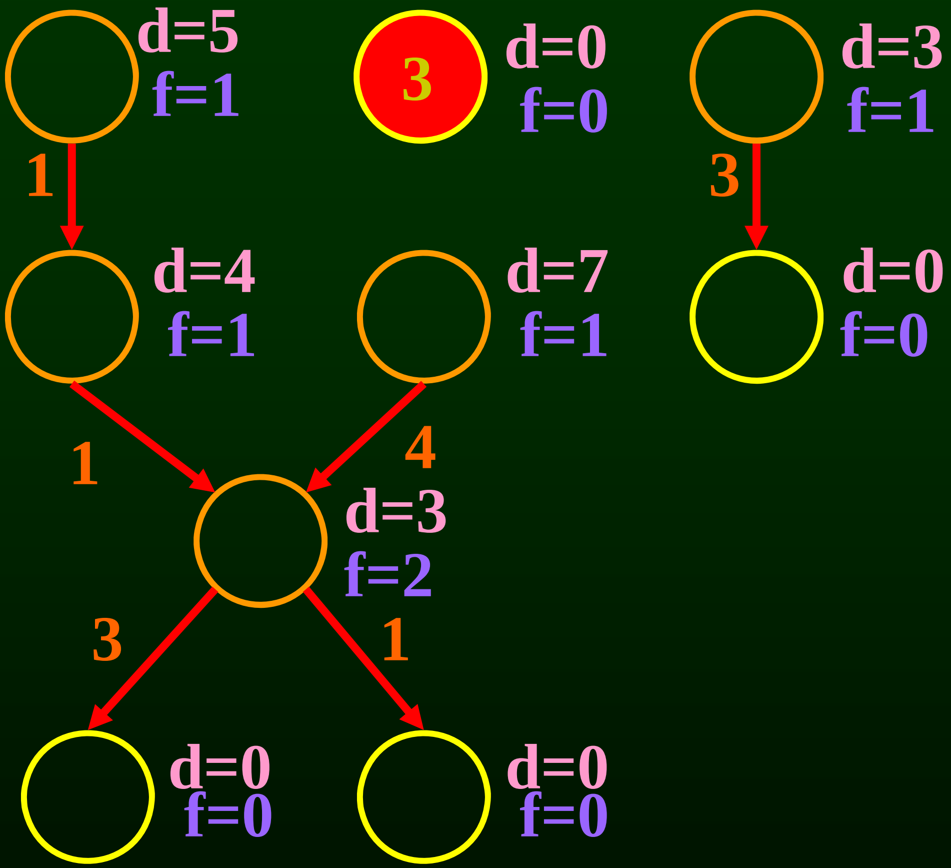
READY = { 7, 3, 5 }





Example

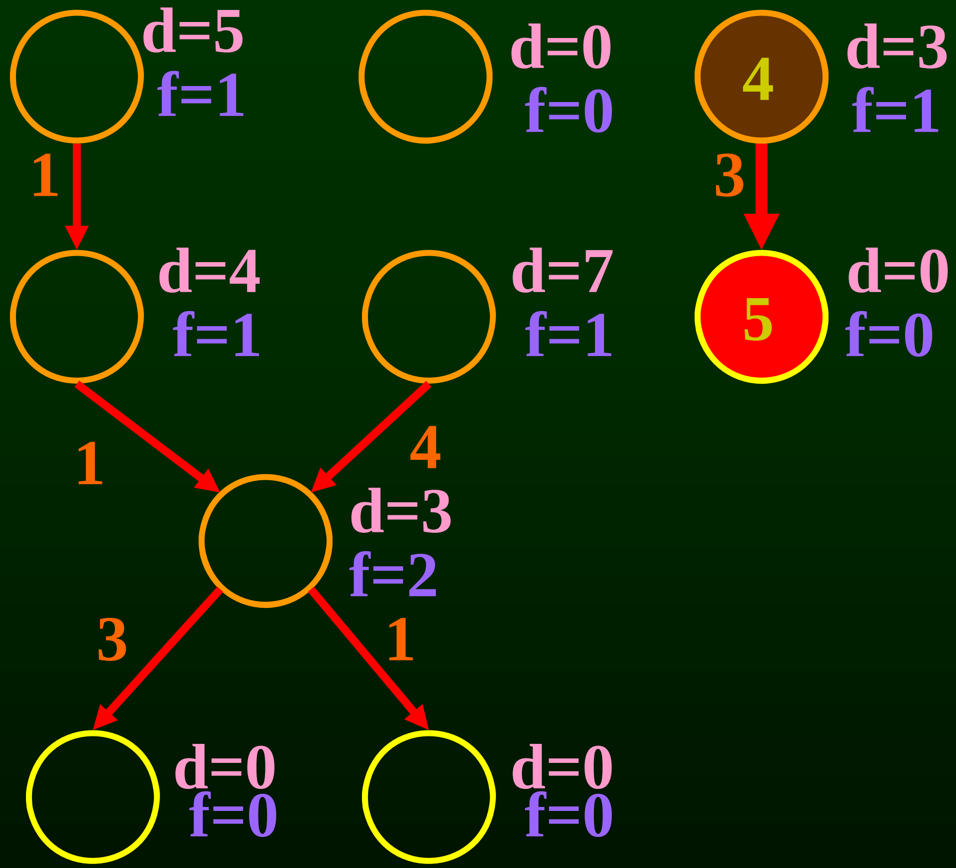
READY = { 3, 5, 8, 9 }

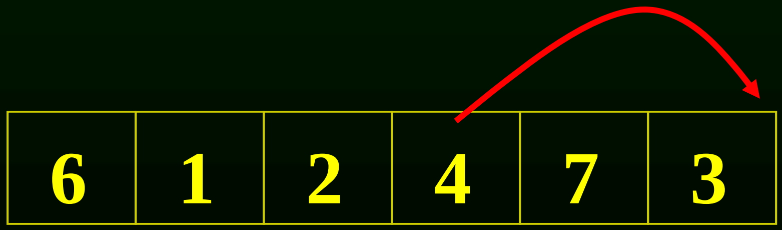


6	1	2	4	7

Example

READY = { 5, 8, 9 }

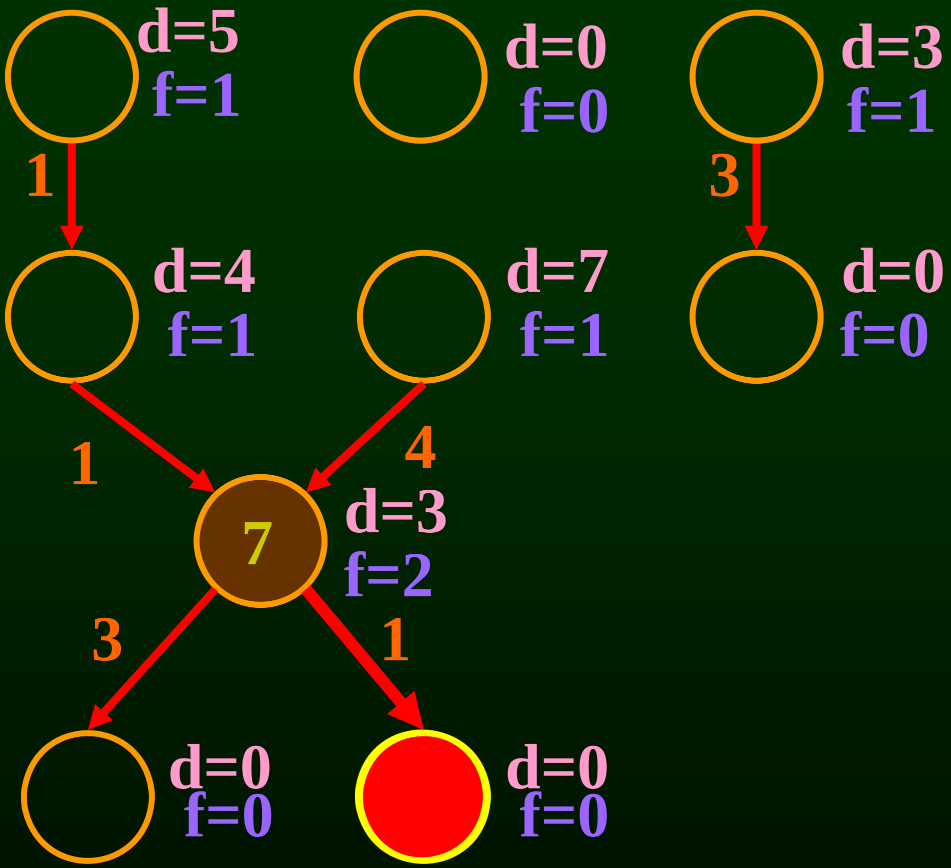


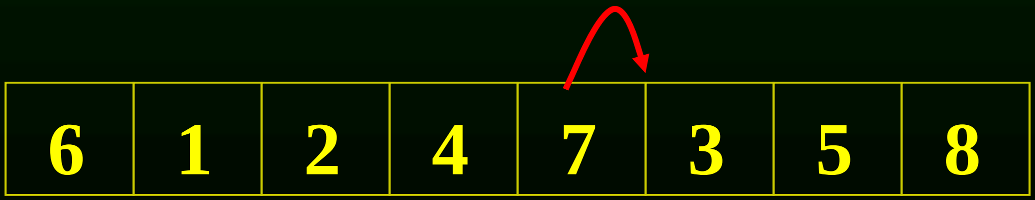


6	1	2	4	7	3

Example

READY = { 9 }

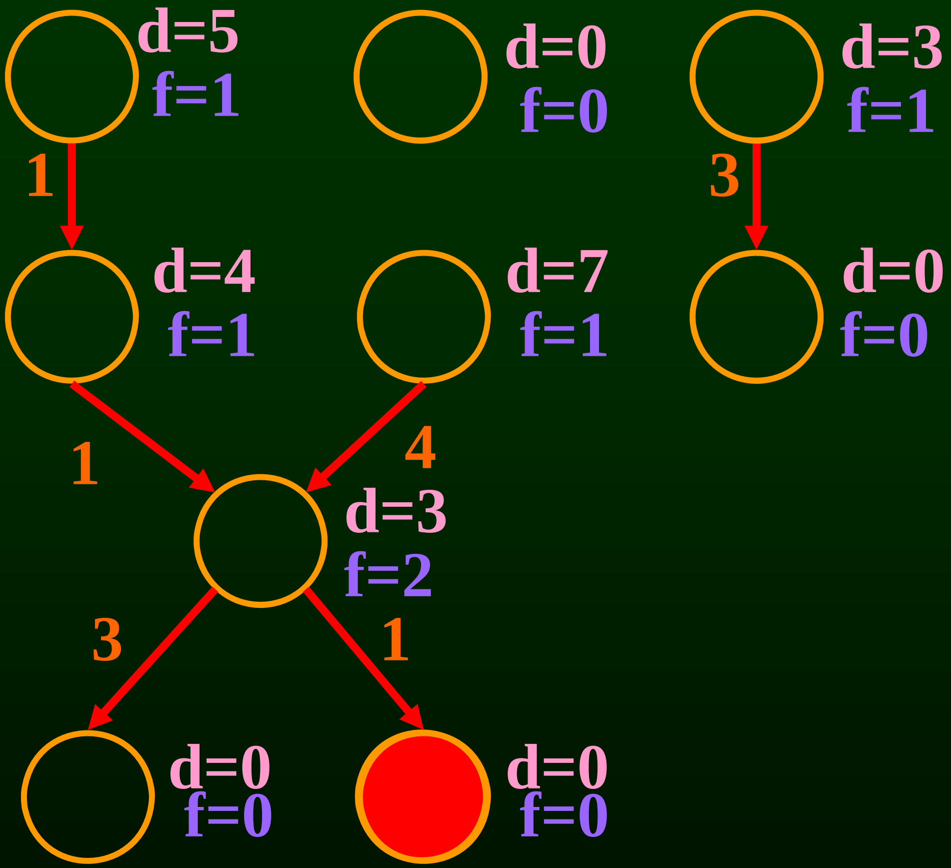




6	1	2	4	7	3	5	8

Example

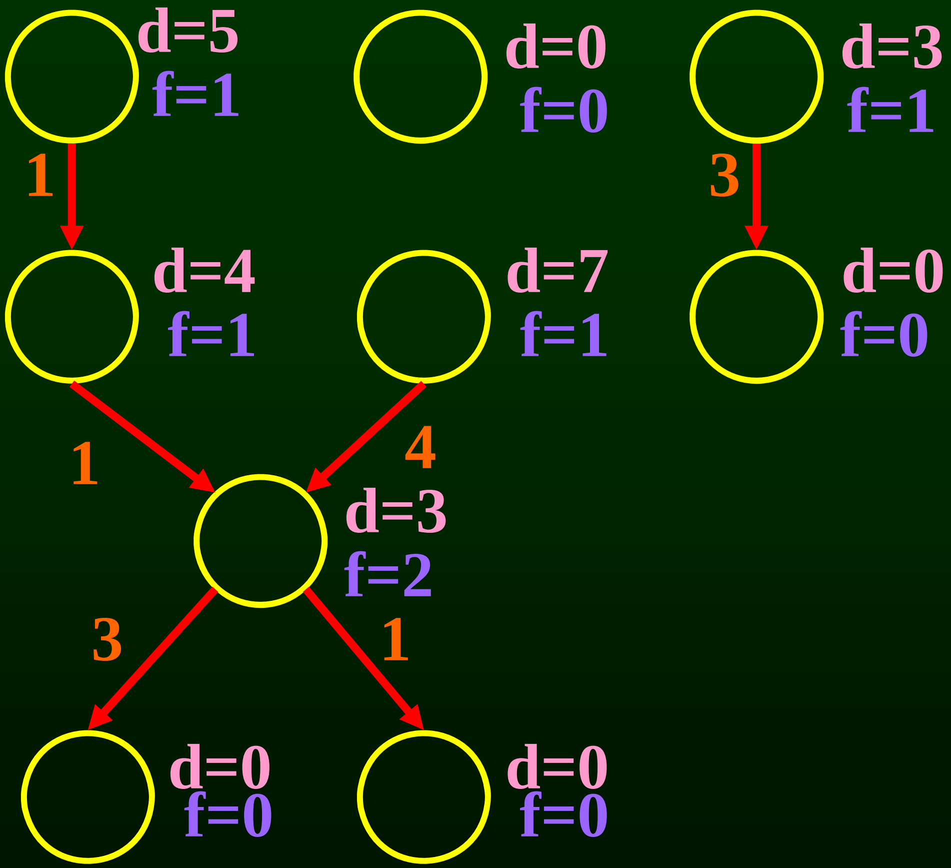
READY = { 9 }



6	1	2	4	7	3	5	8	9

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

Outline

- Modern architectures
- Introduction to instruction scheduling
- List scheduling
- Resource constraints
- Scheduling across basic blocks
- Trace scheduling

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

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

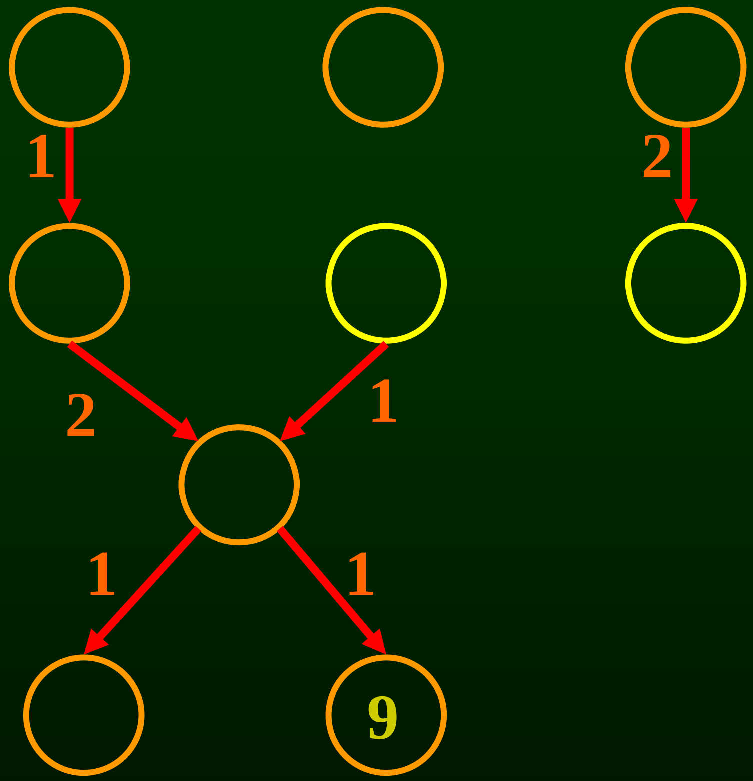
```
READY = nodes with no predecessors
Loop until READY is empty
    Let n $\in$READY be the node with the highest priority
    Schedule n in the earliest slot
        that satisfies precedence + resource constraints
    Update READY
```

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)
```

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				

Outline

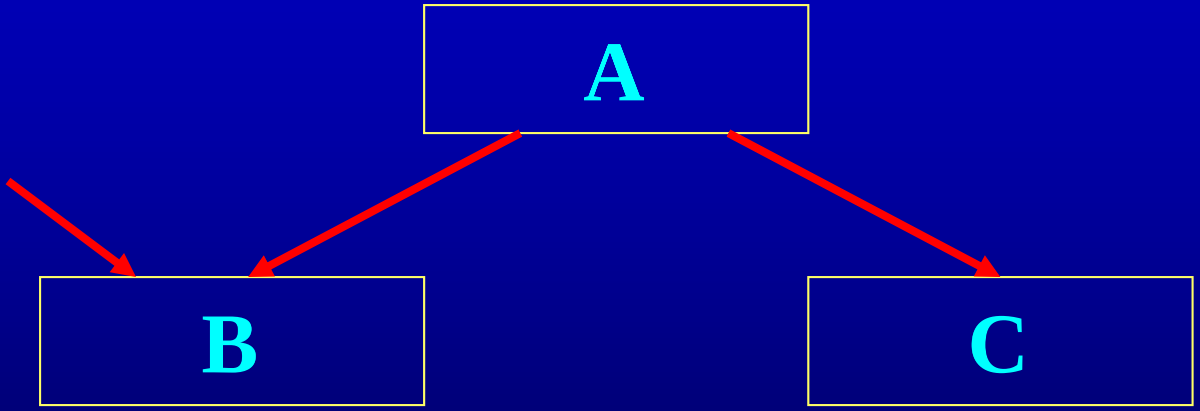
- Modern architectures
- Introduction to instruction scheduling
- List scheduling
- Resource constraints
- Scheduling across basic blocks
- Trace scheduling

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

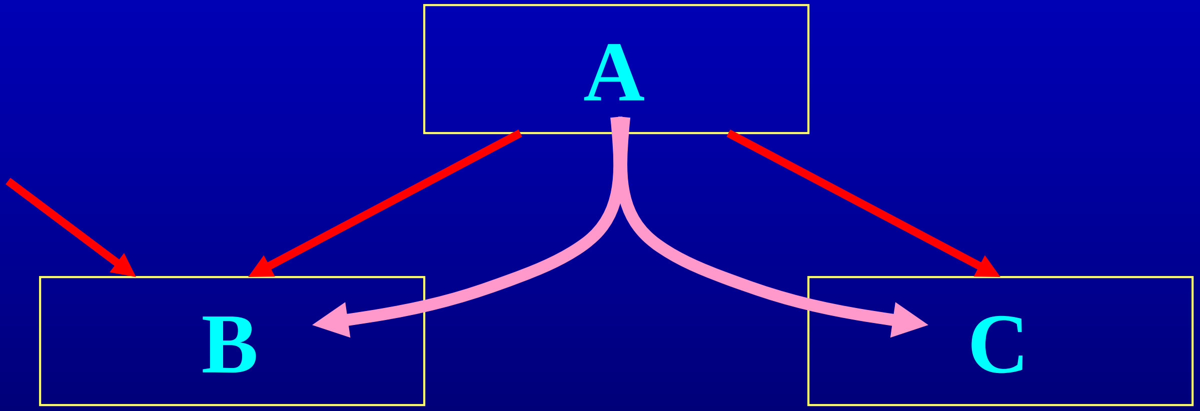
Moving across basic blocks

- Downward to adjacent basic block



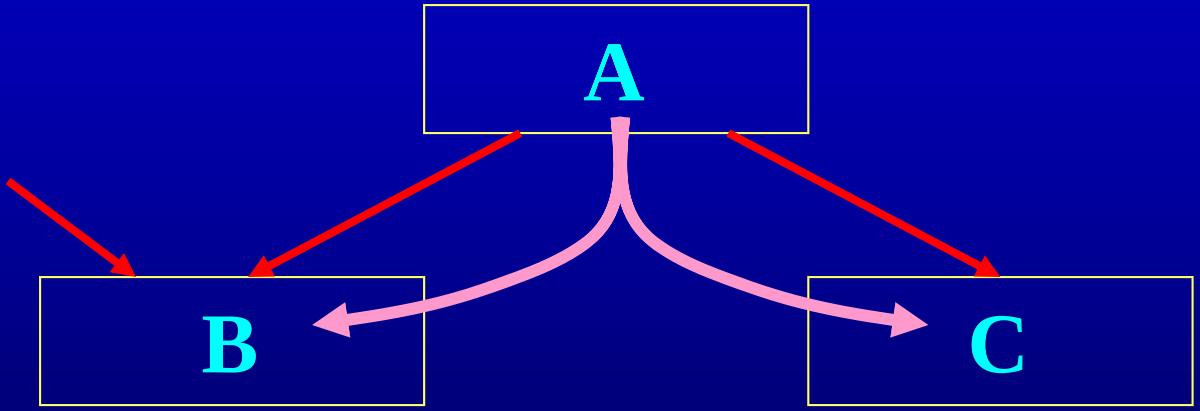
Moving across basic blocks

- Downward to adjacent basic block



Moving across basic blocks

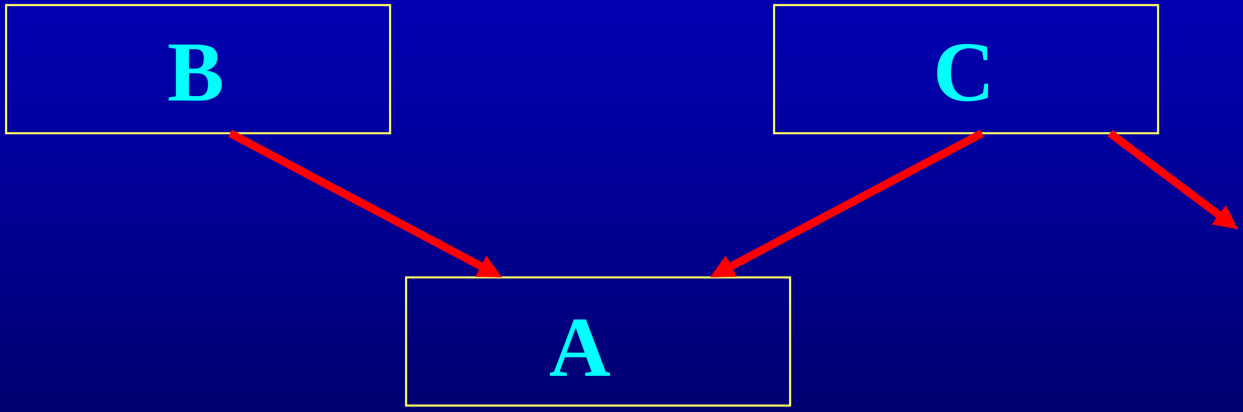
- Downward to adjacent basic block



- A path to B that does not execute A?

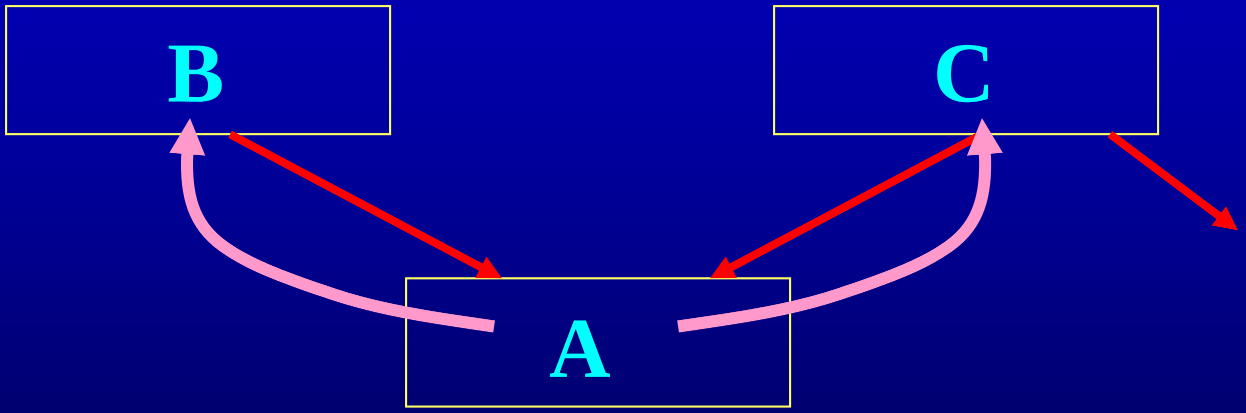
Moving across basic blocks

- Upward to adjacent basic block



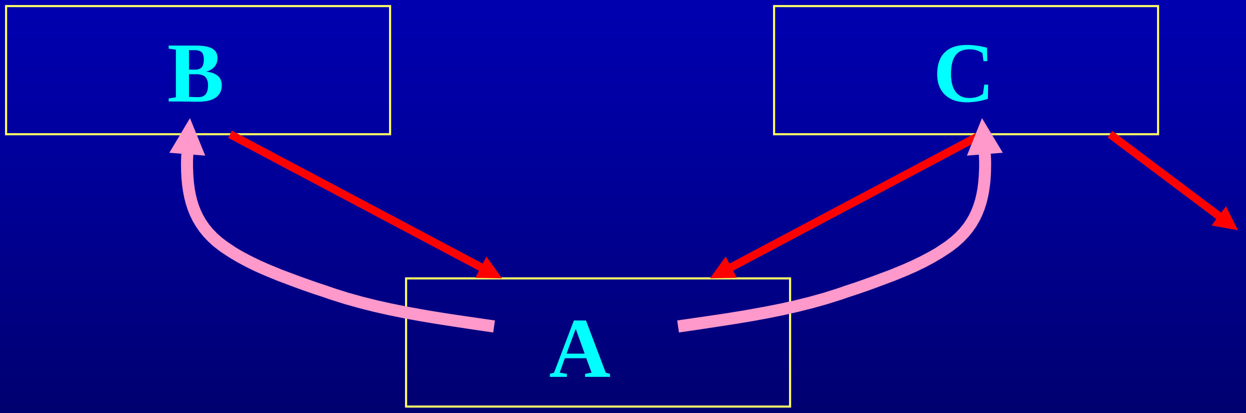
Moving across basic blocks

- Upward to adjacent basic block



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

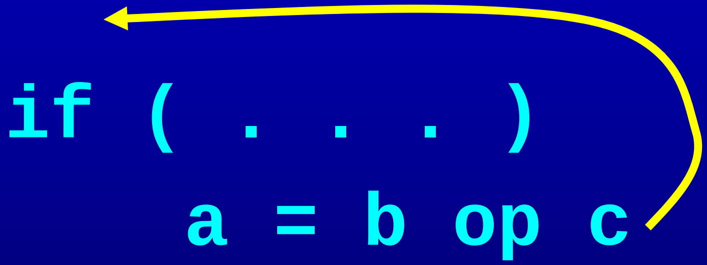
Control Dependencies

- Constraints in moving instructions across basic blocks

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

Control Dependencies

- Constraints in moving instructions across basic blocks



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

Control Dependencies

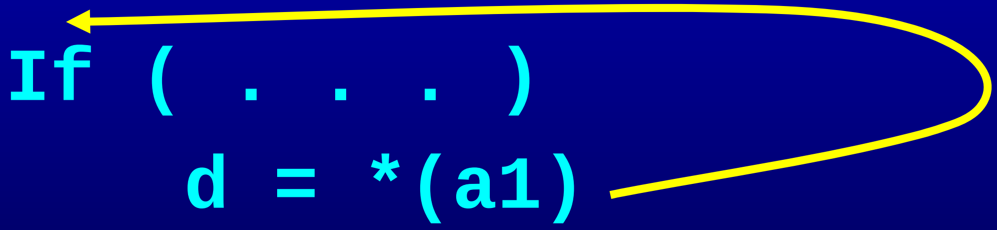
- Constraints in moving instructions across basic blocks

```
if ( c != 0 )
    a = b / c
```

NO!!!

Control Dependencies

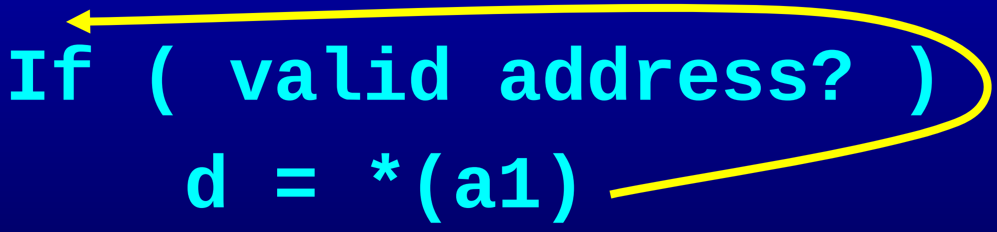
- Constraints in moving instructions across basic blocks



```
If ( . . . )
    d = *(a1)
```

Control Dependencies

- Constraints in moving instructions across basic blocks



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

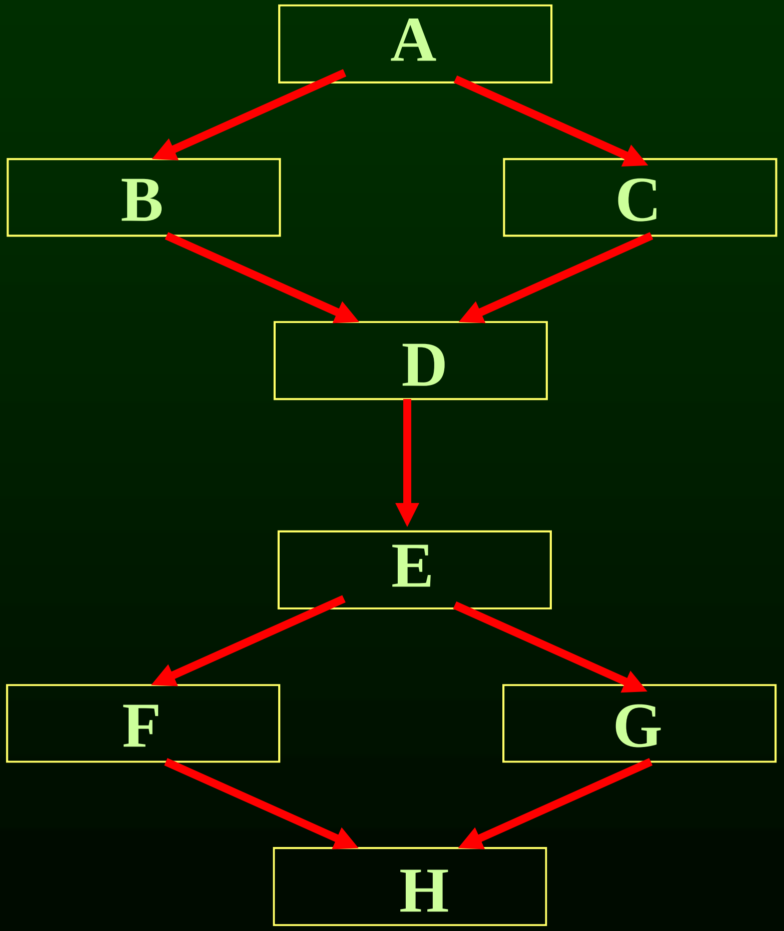
Outline

- Modern architectures
- Introduction to instruction scheduling
- List scheduling
- Resource constraints
- Scheduling across basic blocks
- Trace scheduling

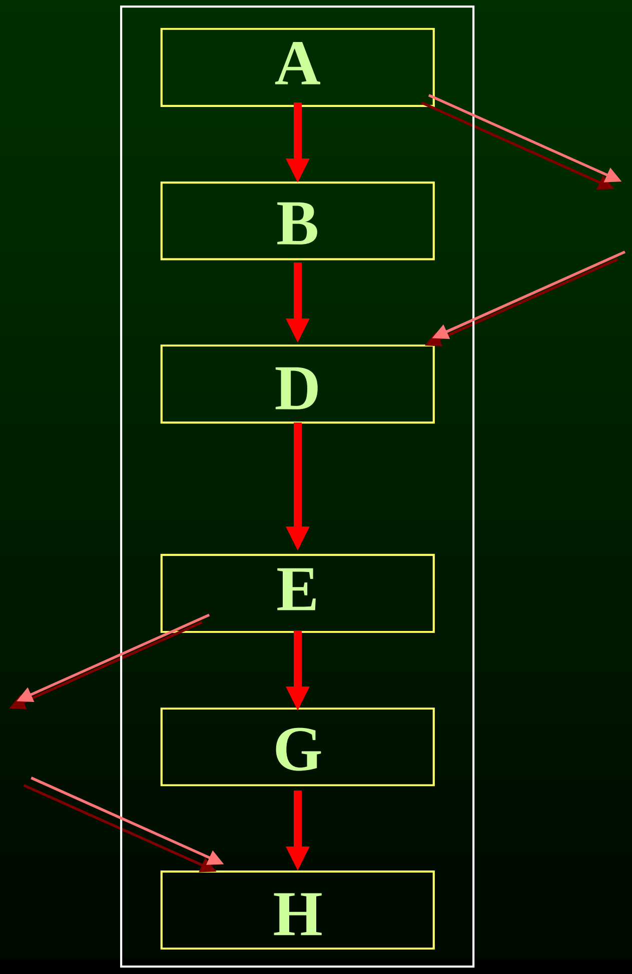
Trace Scheduling

- 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

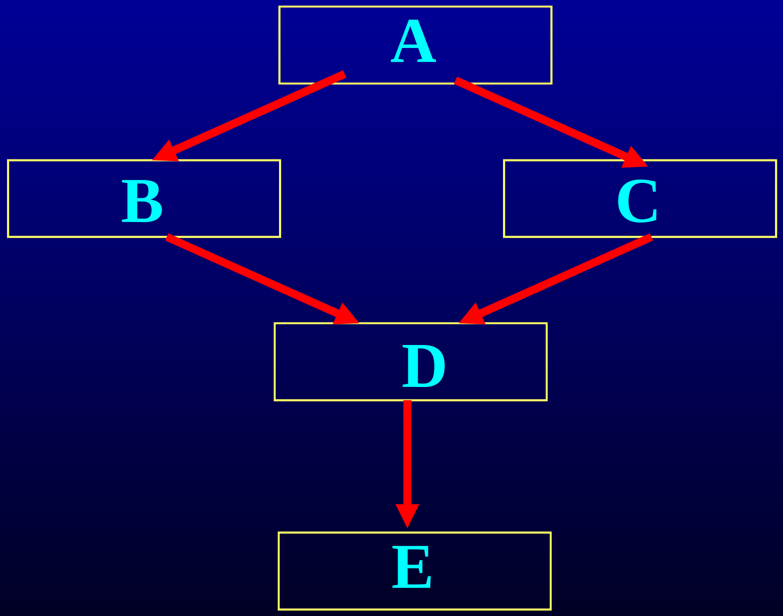

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
B \arrow[dr] & & C \arrow[dl] \\
 & D \arrow[d] & \\
 & E \arrow[dl] \arrow[dr] & \\
F \arrow[dr] & & G \arrow[dl] \\
 & H &
\end{tikzcd}$$

Trace Scheduling



Large Basic Blocks via Code Duplication

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

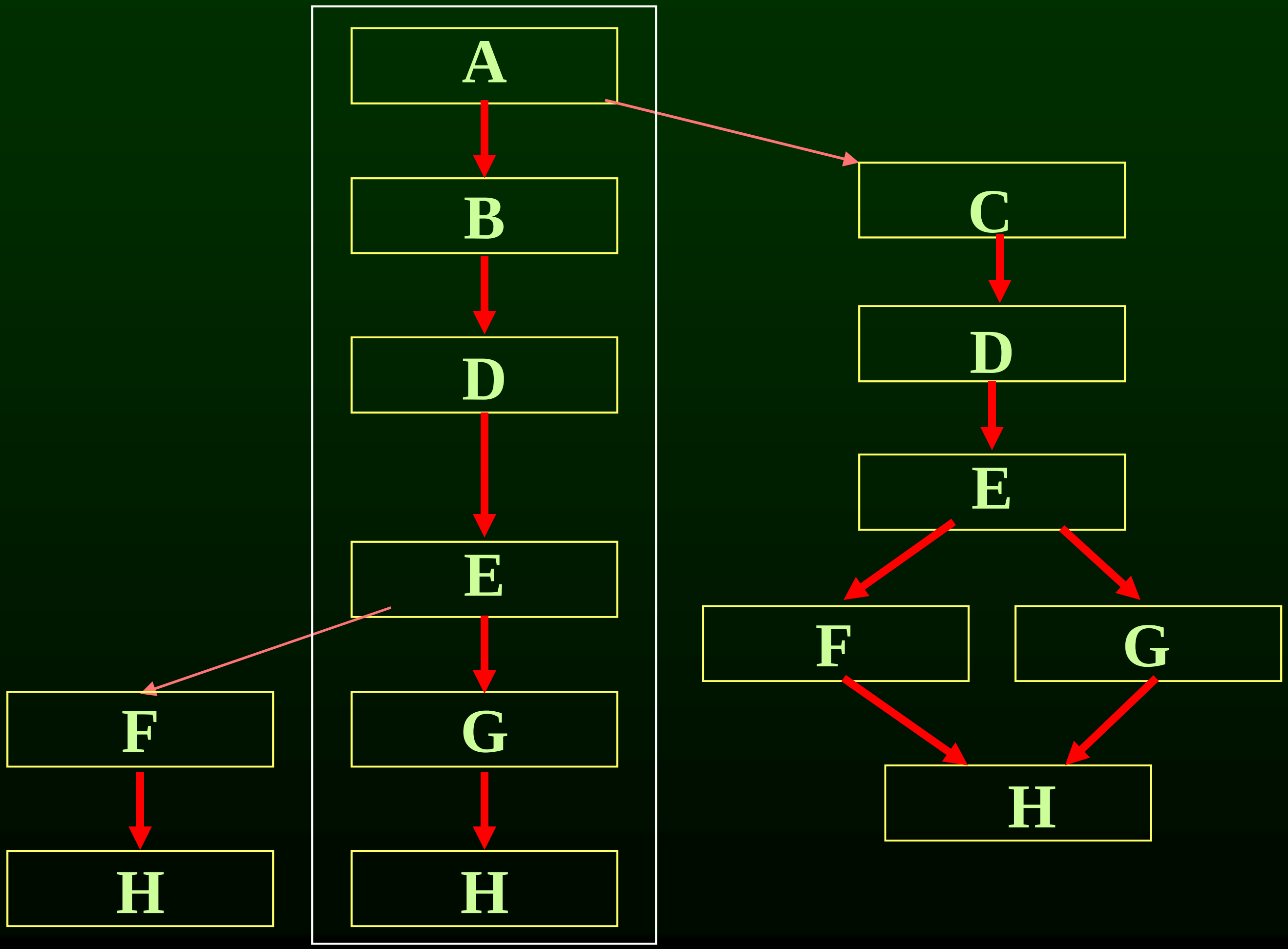

$$\begin{tikzcd} & \boxed{A} \arrow[dl] \arrow[dr] & \\ \boxed{B} \arrow[dr] & & \boxed{C} \arrow[dl] \\ & \boxed{D} \arrow[d] & \\ & \boxed{E} & \end{tikzcd}$$

Large Basic Blocks via Code Duplication

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

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
B \arrow[dr] & & C \arrow[dl] \\
 & D \arrow[d] & \\
 & E &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
B \arrow[d] & & C \arrow[d] \\
D \arrow[d] & & D \arrow[d] \\
E & & E
\end{tikzcd}$$

Trace Scheduling



Next

- Scheduling for loops
- Loop unrolling
- Software pipelining
- Interaction with register allocation
- Hardware vs. Compiler