

# Reading Assignment

- **Read Section 15.4 (Code Scheduling) of *Crafting a Compiler*.**
- **Read Gibbon’s and Muchnick’s paper, “Efficient Instruction Scheduling for a Pipelined Architecture.”**
- **Read Kerns and Eggers’ paper, “Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain.”**

# Code Scheduling

**Modern processors are pipelined.**

**They give the impression that all instructions take unit time by executing instructions in *stages* (steps), as if on an assembly line.**

**Certain instructions though (loads, floating point divides and square roots, delayed branches) take more than one cycle to execute.**

**These instructions may *stall* (halt the processor) or require a nop (null operation) to execute properly.**

**A *Code Scheduling* phase may be needed in a compiler to avoid stalls or eliminate nops.**

# Scheduling Expression DAGs

**After generating code for a DAG or basic block, we may wish to schedule (reorder) instructions to reduce or eliminate stalls.**

***A Postpass Scheduler is run after code selection and register allocation.***

**Postpass schedulers are very general and flexible, since they can be used with code generated by any compiler with any degree of optimization**

***But, since they can't modify register allocations, they can't always avoid stalls.***

# Dependency DAGs

**Obviously, not all reorderings of generated instructions are acceptable.**

**Computation of a register value must precede all uses of that value.**

**A store of a value must precede all loads that might read that value.**

**A *Dependency Dag* reflects essential ordering constraints among instructions:**

- **Nodes are Instructions to be scheduled.**
- **An arc from Instruction *i* to Instruction *j* indicates that *i* must be executed before *j* may be executed.**

# Kinds of Dependencies

**We can identify several kinds of dependencies:**

- **True Dependence:**

**An operation that uses a value has a true dependence (also called a flow dependence) upon an earlier operation that computes the value. For example:**

```
mov    1, %12  
add    %12, 1, %12
```

- **Anti Dependence:**

**An operation that writes a value has a anti dependence upon an earlier operation that reads the value. For example:**

```
add    %12, 1, %10  
mov    1, %12
```

- **Output Dependence:**

**An operation that writes a value has a output dependence upon an earlier operation that writes the value. For example:**

```
mov    1, %12  
mov    2, %12
```

**Collectively, true, anti and output dependencies are called data dependencies. Data dependencies constrain the order in which instructions may be executed.**

# Example

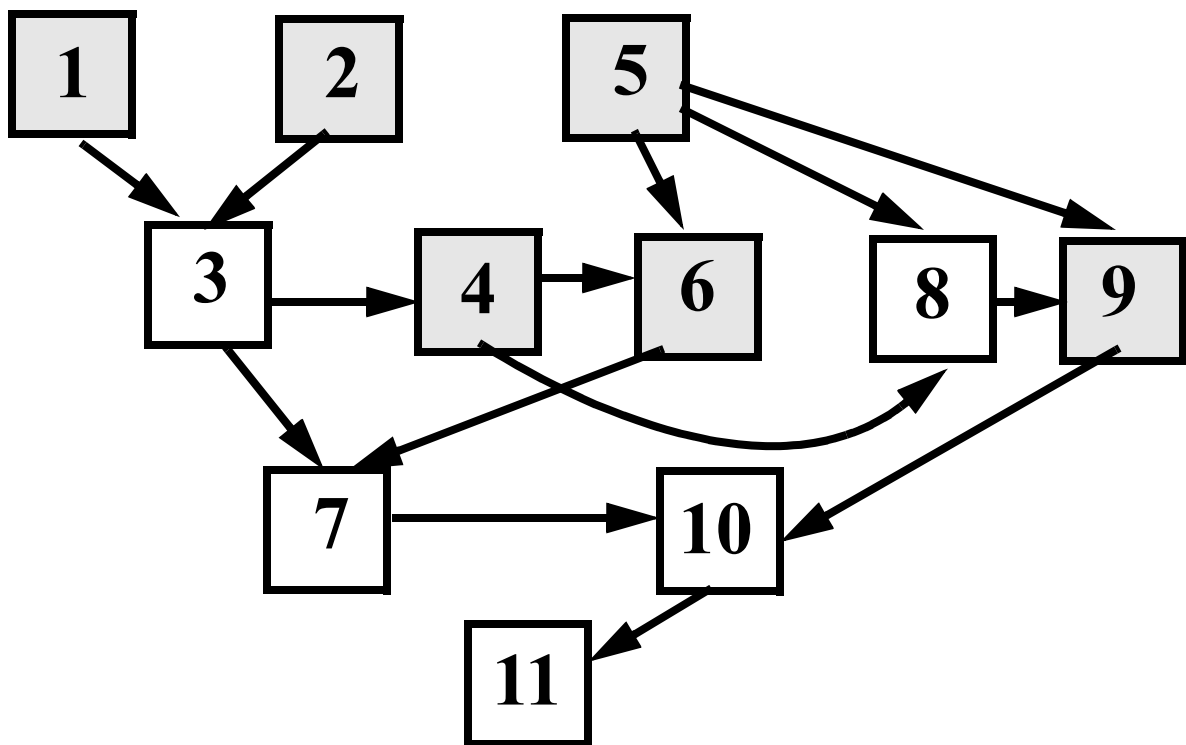
**Consider the code that might be generated for**

**$a = ((a+b) + (c*d)) + ((c+d) * d);$**

**We'll assume 4 registers, the minimum possible, and we'll reuse already loaded values.**

**Assume a 1 cycle stall between a load and use of the loaded value and a 2 cycle stall between a multiplication and first use of the product.**

|         |             |                  |
|---------|-------------|------------------|
| 1. ld   | [a], %r1    |                  |
| 2. ld   | [b], %r2    | ← Stall          |
| 3. add  | %r1,%r2,%r1 |                  |
| 4. ld   | [c], %r2    |                  |
| 5. ld   | [d], %r3    |                  |
| 6. smul | %r2,%r3,%r4 | ← Stall          |
| 7. add  | %r1,%r4,%r1 | ← Stall*2        |
| 8. add  | %r2,%r3,%r2 |                  |
| 9. smul | %r2,%r3,%r2 |                  |
| 10. add | %r1,%r2,%r1 | ← Stall*2        |
| 11. st  | %r1,[a]     | (6 Stalls Total) |





# Scheduling Requires Topological Traversal

**Any valid code schedule is a *Topological Sort* of the dependency dag.**

**To create a code schedule you**

- (1) Pick any root of the Dag.**
- (2) Remove it from the Dag and schedule it.**
- (3) Iterate!**

**Choosing a *Minimum Delay* schedule is NP-Complete:**

**“Computers and Intractability,”  
M. Garey and D. Johnson,  
W.H. Freeman, 1979.**

# Dynamically Scheduled (Out of Order) Processors

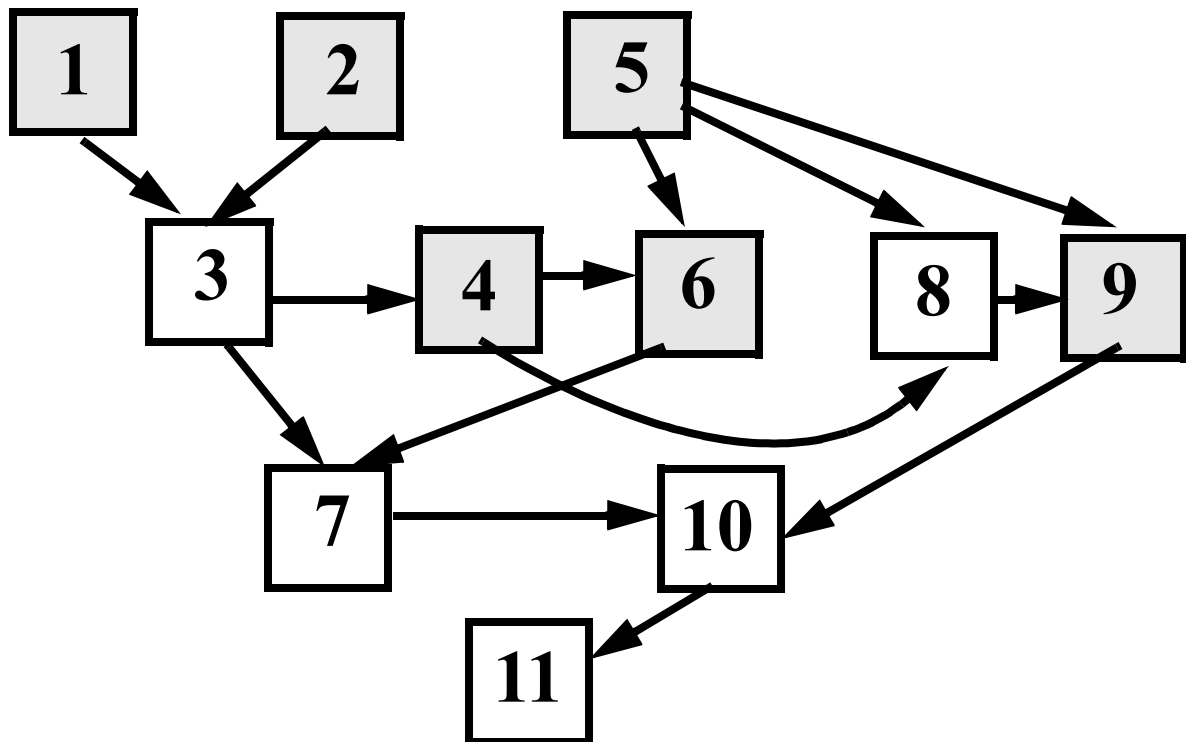
**To avoid stalls, some processors can execute instructions *Out of Program Order*.**

**If an instruction can't execute because a previous instruction it depends upon hasn't completed yet, the instruction can be “held” and a successor instruction executed instead.**

**When needed predecessors have completed, the held instruction is released for execution.**

# Example

```
1. ld    [a], %r1
2. ld    [b], %r2
5. ld    [d], %r3
3. add   %r1, %r2, %r1
4. ld    [c], %r2      ← Stall
6. smul  %r2, %r3, %r4
8. add   %r2, %r3, %r2
9. smul  %r2, %r3, %r2
7. add   %r1, %r4, %r1  ← Stall
10. add  %r1, %r2, %r1
11. st   %r1, [a]      (2 Stalls Total)
```



# Limitations of Dynamic Scheduling

1. Extra processor complexity.
2. Register renaming (to avoid *False Dependencies*) may be needed.
3. Identifying instructions to be delayed takes time.
4. Instructions “late” in the program can’t be started earlier.

# **Gibbons & Muchnick**

## **Postpass Code Scheduler**

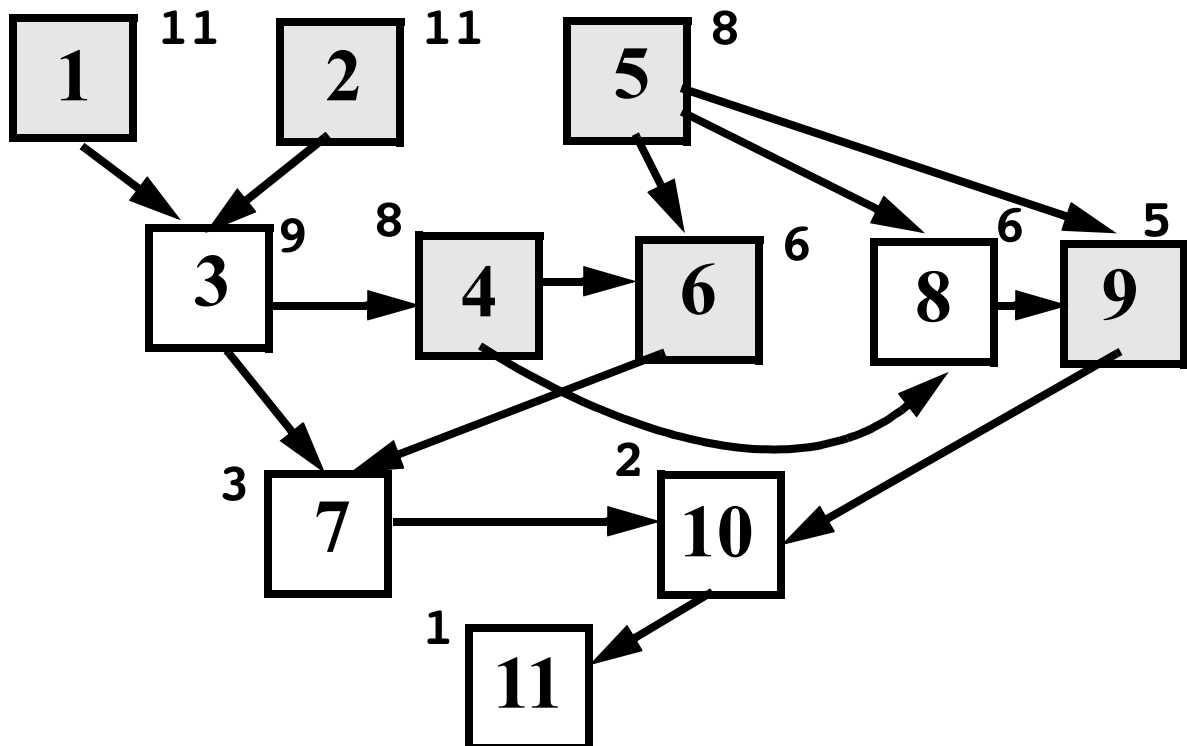
- 1. If there is only one root, schedule it.**
- 2. If there is more than one root, choose that root that won't be stalled by instructions already scheduled.**
- 3. If more than one root can be scheduled without stalling, consider the following rules (in order);**
  - (a) Does this root stall any of its successors?  
(If so, schedule it immediately.)**
  - (b) How many new roots are exposed if this node is scheduled?  
(More is better.)**

**(c) Which root has the longest weighted path to a leaf (using instruction delays as the weight). (The “critical path” in the DAG gets priority.)**

# Example

```

1. ld    [a], %r1 //Longest path
2. ld    [b], %r2 //Exposes a root
5. ld    [d], %r3 //Not delayed
3. add   %r1,%r2,%r1 //Only choice
4. ld    [c], %r2 //Only choice
6. smul  %r2,%r3,%r4 ←//Stalls succ.
8. add   %r2,%r3,%r2 //Not delayed
9. smul  %r2,%r3,%r2 //Not delayed
7. add   %r1,%r4,%r1 //Only choice
10. add  %r1,%r2,%r1 ←//Only choice
11. st   %r1,[a] (2 Stalls Total)
  
```



# False Dependencies

**We still have delays in the schedule that was produced because of “false dependencies.”**

**Both `b` and `c` are loaded into `%r2`. This limits the ability to move the load of `c` prior to any use of `%r2` that uses `b`.**

**To improve our schedule we can use a processor that renames registers *or* allocate additional registers to remove false dependencies.**



# Register Renaming

**Many out of order processors automatically rename distinct uses of the same architectural register to distinct internal registers.**

**Thus**

```
ld  [a],%r1  
ld  [b],%r2  
add %r1,%r2,%r1  
ld  [c],%r2
```

**is executed as if it were**

```
ld  [a],%r1  
ld  [b],%r2  
add %r1,%r2,%r3  
ld  [c],%r4
```

**Now the final load can be executed prior to the add, eliminating a stall.**

# Compiler Renaming

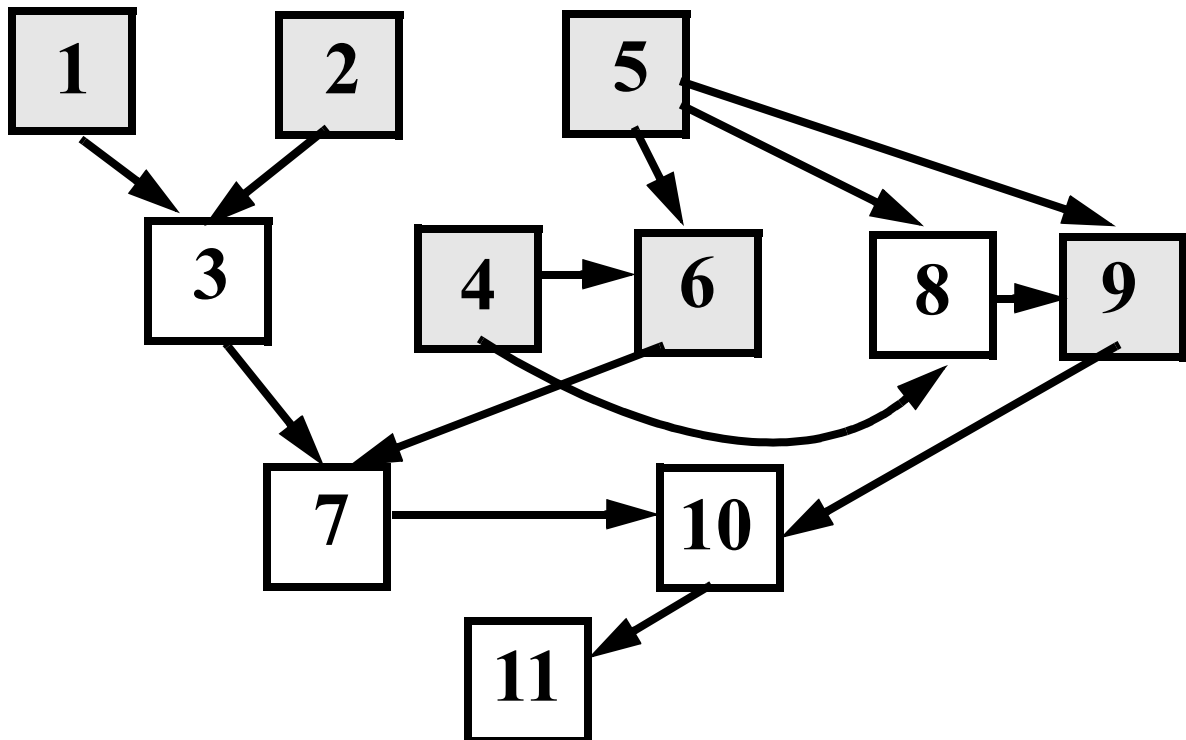
**A compiler can also use the idea of renaming to avoid unnecessary stalls.**

**An extra register may be needed (as was the case for scheduling expression trees).**

**Also, a *round-robin* allocation policy is needed. Registers are reused in a *cyclic* fashion, so that the most recently freed register is reused last, not first.**

# Example

1. ld [a], %r1  
2. ld [b], %r2 ← Stall  
3. add %r1,%r2,%r1  
4. ld [c], %r3  
5. ld [d], %r4  
6. smul %r3,%r4,%r5 ← Stall  
7. add %r1,%r5,%r2 ← Stall\*2  
8. add %r3,%r4,%r3  
9. smul %r3,%r4,%r3  
10. add %r2,%r3,%r2 ← Stall\*2  
11. st %r2,[a] (6 Stalls Total)



# After Scheduling:

```

4. ld    [c], %r3    //Longest path
5. ld    [d], %r4    //Exposes a root
1. ld    [a], %r1    //Stalls succ.
2. ld    [b], %r2    //Exposes a root
6. smul  %r3,%r4,%r5 //Stalls succ.
8. add   %r3,%r4,%r3 //Longest path
9. smul  %r3,%r4,%r3 //Stalls succ.
3. add   %r1,%r2,%r1 //Only choice
7. add   %r1,%r5,%r2 //Only choice
10. add  %r2,%r3,%r2 //Only choice
11. st   %r2,[a]     (0 Stalls Total)
  
```

