Lecture 15

Instruction Scheduling

I. Hardware Support for Parallel Execution
II. Constraints on Scheduling
III. List Scheduling

[ALSU 10.1-10.3]

Optimization: *What's the Point?* (A Quick Review)

Machine-Independent Optimizations:

- e.g., constant propagation & folding, redundancy elimination, dead-code elimination, etc.
- Goal: *eliminate work*

Machine-Dependent Optimizations:

- register allocation
  - Goal: *reduce cost of accessing data*
- instruction scheduling
  - Goal: *???*
- …

The Goal of Instruction Scheduling

- Assume that the remaining instructions are all essential
  - (otherwise, earlier passes would have eliminated them)
- How can we perform this fixed amount of work in less time?
  - Answer: *execute the instructions in parallel*

Time

```
a = 1 + x;
b = 2 + y;
c = 3 + z;
```

```
a = 1 + x; b = 2 + y; c = 3 + z;
```


I. Hardware Support for Parallel Execution

- Three forms of parallelism are found in modern machines:
  - Pipelining
  - Superscalar Processing
  - Multicore

Instruction Scheduling (Pipelining, Superscalar Processing)

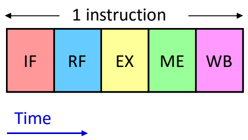
Automatic Parallelization (Multicore)
[future lecture]

Pipelining

Basic idea:

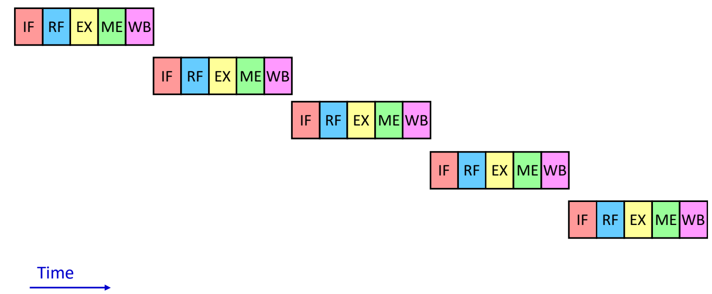
– break instruction into *stages* that can be overlapped

Example: simple 5-stage pipeline from early RISC machines

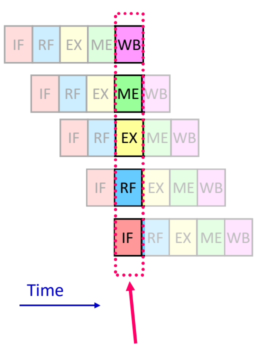


IF = Instruction Fetch
RF = Decode & Register Fetch
EX = Execute on ALU
ME = Memory Access
WB = Write Back to Register File

Pipelining Illustration



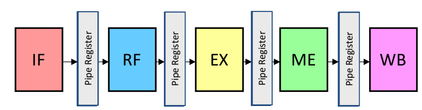
Pipelining Illustration



- In a given cycle, each instruction is in a different stage

Beyond 5-Stage Pipelines: Even More Parallelism

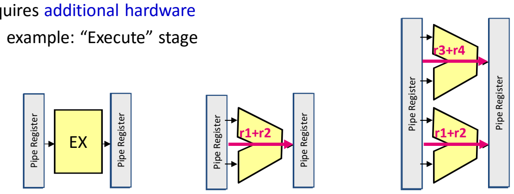
- Should we simply make pipelines deeper and deeper?



  - registers between pipeline stages have fixed overheads
    - hence diminishing returns with more stages (Amdahl's Law)
  - value of pipe stage unclear if < time for integer add
- However, many consumers think "performance = clock rate"
  - perceived need for higher clock rates -> deeper pipelines
  - e.g., Pentium 4 processor had a 20-stage pipeline [2000-2008]

Beyond Pipelining: "Superscalar" Processing

- Basic Idea:
  - multiple (independent) instructions can proceed simultaneously through the same pipeline stages
- Requires additional hardware
  - example: "Execute" stage

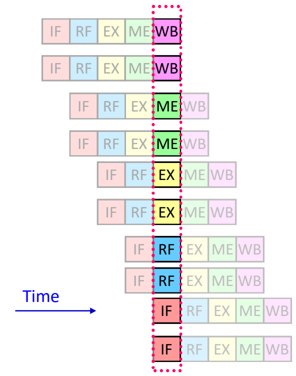


Abstract Representation

Hardware for Scalar Pipeline: 1 ALU

Hardware for 2-way Superscalar: 2 ALUs

Superscalar Pipeline Illustration



Original (scalar) pipeline:

- Only one instruction in a given pipe stage at a given time

Superscalar pipeline:

- Multiple instructions in the same pipe stage at the same time

II. Constraints on Scheduling

1. Hardware Resources
2. Data Dependences
3. Control Dependences

Constraint #1: Hardware Resources

- Processors have finite resources, and there are often constraints on how these resources can be used.

Examples:

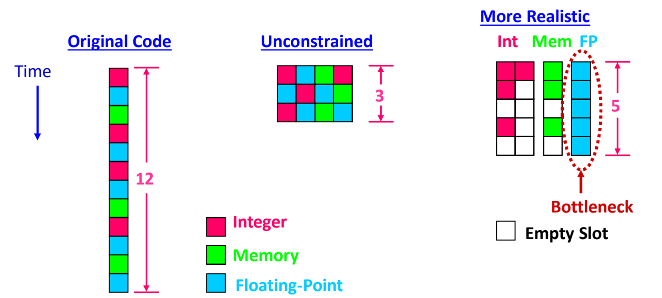
- Finite issue width
- Limited functional units (FUs) per given instruction type
- Limited pipelining within a given functional unit (FU)

Finite Issue Width

- Prior to superscalar processing:
  - processors only “issued” one instruction per cycle
- Even with superscalar processing:
  - limit on total # of instructions issued per cycle

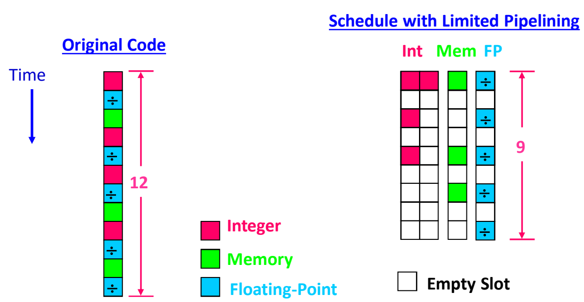
Limited FUs per Instruction Type

- e.g., a 4-way superscalar might only be able to issue up to 2 integer, 1 memory, and 1 floating-point insts per cycle



Limited Pipelining within a Functional Unit

- e.g., only 1 new floating-point division once every 2 cycles



Constraints on Scheduling

1. Hardware Resources
2. Data Dependences
3. Control Dependences

Constraint #2: Data Dependences

- If we read or write a data location "too early", the program may behave incorrectly.

(Assume that initially, `x = 0`*.)*

???
```
x = 1;
y = x;
```
Read-after-Write
("True" dependence)

Fundamental
(no simple fix)

???
```
x = 1;
x1= 2;
```
Write-after-Write
("Output" dependence)

???
```
y = x;
x1= 1;
```
Write-after-Read
("Anti" dependence)

Can potentially fix through *renaming*.

Why Data Dependences are Challenging

```
 x = a[i];
*p = 1;
 y = *q;
*r = z;
```

- which of these instructions can be reordered?
- *ambiguous data dependences* are very common in practice
  - difficult to resolve, despite fancy pointer analysis [next lecture]

Given Ambiguous Data Dependences, What To Do?

```
 x = a[i];
*p = 1;
 y = *q;
*r = z;
```

- Conservative approach: don't reorder instructions
  - ensures correct execution
  - but may suffer poor performance
- Aggressive approach?
  - is there a way to safely reorder instructions?

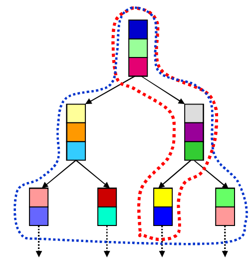
Hardware Limitations: Multi-cycle Execution Latencies

- Simple instructions often "execute" in one cycle
  - (as observed by other instructions in the pipeline)
  - e.g., integer addition
- More complex instructions may require multiple cycles
  - e.g., integer division, square-root
  - cache misses!
- These latencies, when combined with data dependencies, can result in non-trivial critical path lengths through code

Constraints on Scheduling

1. Hardware Resources
2. Data Dependences
3. Control Dependences

Constraint #3: Control Dependences



- What do we do when we reach a conditional branch?
  - choose a "frequently-executed" path?
  - choose multiple paths?

Scheduling Constraints: Summary

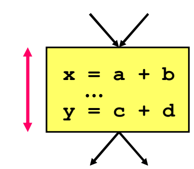
- Hardware Resources
  - finite set of FUs with instruction type, bandwidth, and latency constraints
  - cache hierarchy also has many constraints
- Data Dependences
  - can't consume a result before it is produced
  - ambiguous dependences create many challenges
- Control Dependences
  - impractical to schedule for all possible paths
  - choosing an "expected" path may be difficult
    - recovery costs can be non-trivial if you are wrong

III. List Scheduling

- The most common technique for scheduling instructions within a basic block

Basic block scheduling doesn't need to worry about:
- control flow [future lecture]

Does need to worry about:
- data dependences
- hardware resources



- Even without control flow, the problem is still NP-hard

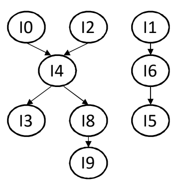
List Scheduling Algorithm: Inputs and Outputs

Algorithm reproduced from:

– *"An Experimental Evaluation of List Scheduling"*, Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Rice University, Department of Computer Science Technical Report 98-326, September 1998.

Inputs:

Data Precedence Graph (DPG)



Machine Parameters

# of FUs:
2 INT, 1 FP
Latencies:
add = 1 cycle, …
Pipelining:
1 add/cycle, …

Output:

Scheduled Code

ALU 0	ALU 1	FP	Cycle
I0	I2	---	0
---	I1	I4	1
I3	I8	I6	2
I10	---	I11	3
I7	I9	I5	4

(I7, I10, I11 not shown in DPG)

List Scheduling: The Basic Idea

- Maintain a list of instructions that are ready to execute
  - data dependence constraints would be preserved
  - machine resources are available
- Moving cycle-by-cycle through the schedule template:
  - choose instructions from the list & schedule them
  - update the list for the next cycle

I2 I0

			Cycle
		---	0 ←
			1
			2

What Makes Life Interesting: Choice

Easy case:

– all ready instructions can be scheduled this cycle

I5 I1 I7

			←

Interesting case:

– we need to pick a subset of the ready instructions

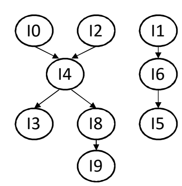
I5 I1 I0 I2 I7 ???

			←

- List scheduling makes choices based upon *priorities*
  - assigning priorities correctly is a key challenge

Intuition Behind Priorities

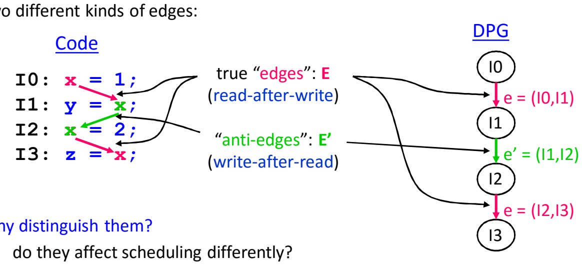
- Intuitively, what should the priority correspond to?
- What factors are used to compute it?
  - data dependences?
  - machine parameters?



# of FUs:
2 INT, 1 FP
Latencies:
add = 1 cycle, …
Pipelining:
1 add/cycle, …

Representing Data Dependences: The Data Precedence Graph (DPG)

- Two different kinds of edges:



- Why distinguish them?
  - do they affect scheduling differently?

    RAW: read waits for value to be computed

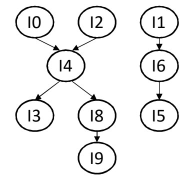
    WAR: write only needs to wait for read to start
- What about output dependences?

    WAW: earlier write is removed by Dead Code Elimination

Computing Priorities

- Let's start with just true dependences (i.e. "edges" in DPG)
- Priority = *latency-weighted depth* in the DPG

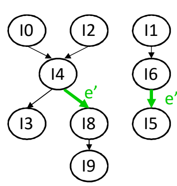
$$priority(x) = max(\forall_{l \in leaves(DPG)} \forall_{p \in paths(x,\ldots,l)} \sum_{p_i=x}^{l} latency(p_i))$$



Computing Priorities (Cont.)

- Now let's also take anti-dependences into account
  - i.e. anti-edges in the set E'

$$priority(x) = \begin{cases} latency(x) & \text{if } x \text{ is a leaf} \\ max(latency(x) + max_{(x,y) \in E}(priority(y)), & \\ \quad max_{(x,y) \in E'}(priority(y))) & otherwise. \end{cases}$$



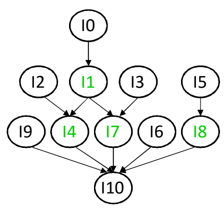
List Scheduling Algorithm

```
cycle = 0;
ready-list = root nodes in DPG; inflight-list = {};

while ((|ready-list|+|inflight-list| > 0) && an issue slot is available) {
    for op = (all nodes in ready-list in descending priority order) {
        if (an FU exists for op to start at cycle) {
            remove op from ready-list and add to inflight-list;
            add op to schedule at time cycle;
            if (op has an outgoing anti-edge)
                add all targets of op's anti-edges that are ready to ready-list;
        }
    }
    cycle = cycle + 1;
    for op = (all nodes in inflight-list)
        if (op finishes at time cycle) {
            remove op from inflight-list;
            check nodes waiting for op & add to ready-list if all operands available;
        }
    }
}
```

List Scheduling Example

```
I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19;
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1
```

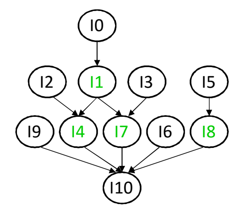


		Cycle
		0
		1
		2
		3
		4
		5
		6

- 2 identical fully-pipelined FUs
- adds take 2 cycles; all other insts take 1 cycle

List Scheduling Example

```
I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19;
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1
```

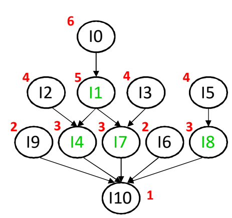


		Cycle
I0	I2	0
I1	I3	1
I5	I9	2
I4	I7	3
I8	I6	4
---	---	5
I10		6

- 2 identical fully-pipelined FUs
- adds take 2 cycles; all other insts take 1 cycle

What if We Break Ties Differently?

```
I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19;
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1
```

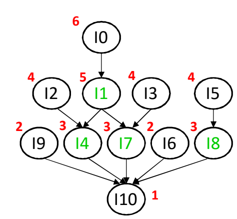


		Cycle
I0	I2	0
I1	I5	1
I3	I8	2
I4	I7	3
I9	I6	4
I10		5
		6

- 2 identical fully-pipelined FUs
- adds take 2 cycles; all other insts take 1 cycle

Contrasting the Two Schedules

- Breaking ties arbitrarily may not be the best approach

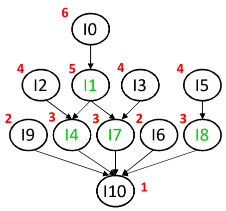


		Cycle
I0	I2	0
I1	I3	1
I5	I9	2
I4	I7	3
I8	I6	4
---	---	5
I10		6

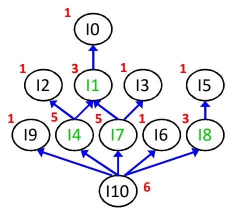
		Cycle
I0	I2	0
I1	I5	1
I3	I8	2
I4	I7	3
I9	I6	4
I10		5

Backward List Scheduling

Modify the algorithm as follows:
- reverse the direction of all edges in the DPG
- schedule the *finish times* of each operation
  - start times must still be used to ensure FU availability



Forward Scheduling Priorities



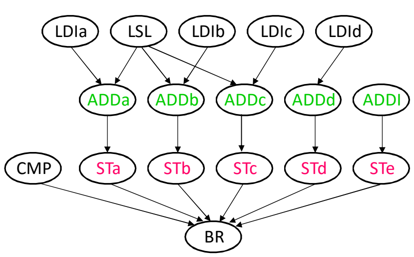
Backward Scheduling Priorities

Backward List Scheduling

Modify the algorithm as follows:
- reverse the direction of all edges in the DPG
- schedule the *finish times* of each operation
  - start times must still be used to ensure FU availability

Impact of scheduling backwards:
- clusters operations near the end (vs. the beginning)
- may be either better or worse than forward scheduling

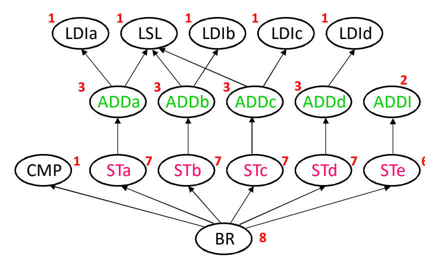
Backward List Scheduling Example: Let's Schedule it Forward First



INT	INT	MEM	Cycle
LDIa	LSL	----	0
LDIb	LDIc	----	1
LDId	ADDa	----	2
ADDb	ADDc	----	3
ADDd	ADDI	STa	4
CMP	----	STb	5
----	----	STc	6
----	----	STd	7
----	----	STe	8
----	----	----	9
----	----	----	10
----	----	----	11
BR	----	----	12

Hardware parameters:
- 2 INT units: ADDs take 2 cycles; others take 1 cycle
- 1 MEM unit: stores (ST) take 4 cycles

Now Let's Try Scheduling Backward



INT	INT	MEM	Cycle
LDIa	----	----	0
ADDI	LSL	----	1
ADDd	LDIc	----	2
ADDc	LDId	STe	3
ADDb	LDIa	STd	4
ADDa	----	STc	5
----	----	STb	6
----	----	STa	7
----	----	----	8
----	----	----	9
CMP	----	----	10
BR	----	----	11

Hardware parameters:
- 2 INT units: ADDs take 2 cycles; others take 1 cycle
- 1 MEM unit: stores (ST) take 4 cycles

Contrasting Forward vs. Backward List Scheduling

Forward

INT	INT	MEM	Cycle
LDIa	LSL	----	0
LDIb	LDIc	----	1
LDId	ADDa	----	2
ADDb	ADDc	----	3
ADDd	ADDI	STa	4
CMP	----	STb	5
----	----	STc	6
----	----	STd	7
----	----	STe	8
----	----	----	9
----	----	----	10
----	----	----	11
BR	----	----	**12**

Backward

INT	INT	MEM	Cycle
LDIa	----	----	0
ADDI	LSL	----	1
ADDd	LDIc	----	2
ADDc	LDId	STe	3
ADDb	LDIa	STd	4
ADDa	----	STc	5
----	----	STb	6
----	----	STa	7
----	----	----	8
----	----	----	9
CMP	----	----	10
BR	----	----	**11**

- backward scheduling clusters work near the end
- backward is better in this case, but this is not always true

Evaluation of List Scheduling

Cooper *et al.* propose "RBF" scheduling:
- schedule each block M times forward & backward
- break any priority ties randomly

For real programs:
- regular list scheduling works very well

For synthetic blocks:
- RBF wins when "available parallelism" (AP) is ~2.5
- for smaller AP, scheduling is too constrained
- for larger AP, any decision tends to work well

List Scheduling Wrap-Up

- The priority function can be arbitrarily sophisticated
  - e.g., filling branch delay slots in early RISC processors
- List scheduling is widely used, and it works fairly well
- It is limited, however, by basic block boundaries

Efficient Instruction Scheduling for a Pipelined Architecture

Phillip B. Gibbons* & Steven S. Muchnick**

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304-1181

[My first publication. "PLDI" 1986]

Abstract

As part of an effort to develop an optimizing compiler for a pipelined architecture, a code reorganization algorithm has been developed that significantly reduces the number of run-time pipeline interlocks. In a pass after code generation, the algorithm uses a dag representation to heuristically schedule the instructions in each basic block.

Previous algorithms for reducing pipeline interlocks have had worst-case runtimes of at least $O(n^4)$. By using a dag representation which prevents scheduling deadlocks and a selection method that requires no lookahead, the resulting algorithm reorganizes instructions almost as effectively in practice, while having an $O(n^2)$ worst-case runtime.

1. Introduction

The architecture we have studied has many features which

Fortunately, not all pairs of consecutive instructions cause pipeline hazards. In the architecture under consideration, the only hazards are register- and memory-based: 1) loading a register from memory followed by using *that* register as a source, 2) storing to any memory location followed by loading from any location, and 3) loading from memory followed by using *any* register as the target of an arithmetic/logical instruction or a load/store with address modification. Each of these pipeline hazards causes some potential implementation of the architecture to stall or *interlock* for one pipe cycle.

There are three approaches to reducing the number of pipeline interlocks incurred in executing a program, distinguished by the agent and the time when the code is inspected: either special hardware can do it during execution, or a person or software can do it before execution. The hardware approach has been used in the Control Data 6600 [Tho64] and the IBM 360/91 [Tom67], two of the fastest machines of their day. While reasonably effective, this approach is very expensive and can only span relatively short code sequences. Rymarczyk

Looking Ahead

- Monday: Pointer Analysis [ALSU 12.4, 12.6-12.7]
- Wednesday: Dynamic Code Optimization
- Friday: No class
- Following Monday & Wednesday: “Recent Research on Optimization”
  - Student-led discussions, in groups of 2, with 20 minutes/group
  - Read 3 papers on a topic, and lead a discussion in class