CS107: Instruction Selection & Scheduling

Guannan Wei
guannan.wei@tufts.edu

April 16, 2026

Spring 2026

Tufts University

Today

Backend architecture-dependent optimizations:

- Instruction selection: *what* are the machine instructions we should generate
- Instruction scheduling: *when* should we execute these instructions

Instruction selection

Directly translating low-level IR into machine instructions is straightforward, for example, mapping RTL code of array indexing `x <- a[i]` to assembly:

```
movq i, %rax
imulq $8, %rax
addq a, %rax
movq (%rax), %rax
```

Instruction selection

Directly translating low-level IR into machine instructions is straightforward, for example, mapping RTL code of array indexing `x <- a[i]` to assembly:

```
movq i, %rax
imulq $8, %rax
addq a, %rax
movq (%rax), %rax
```

But using X86 addressing mode, a faster one would be

```
movq (a, i, 8), %rax
```

Given the knowledge of the target machine's instruction set, how do we map computation to hardware efficiently?

Instruction selection

Mapping multiple low-level IR instructions:

```
t1 <- a + b
t2 <- t1 * 4
```

Naive translation:

```
mov a, %rax
add b, %rax
imul 4, %rax
```

Instruction selection

Mapping multiple low-level IR instructions:

```
t1 <- a + b
t2 <- t1 * 4
```

Naive translation:

```
mov a, %rax
add b, %rax
imul 4, %rax
```

Better translation using `lea` (load effective address) and shift:

```
lea (a, b), %rax
shl $2, %rax
```

Instruction selection

Instruction selection: choosing a particular instruction sequence to implement the corresponding low-level IR (e.g. RTL) code, among many possible ones.

Highly target machine dependent, especially useful for CISC architectures.

Instruction selection

Instruction selection: choosing a particular instruction sequence to implement the corresponding low-level IR (e.g. RTL) code, among many possible ones.

Highly target machine dependent, especially useful for CISC architectures.

Need to consider the cost depending on the architecture:

- Temporal cost: cycles of instructions
- Spatial cost: size of the generated code

Instruction selection

Solution: Represent the low-level IR as a tree or DAG, and match it with patterns (subtrees, or tiles), where each pattern can be mapped to machine instructions. Different algorithms exist:

- Maximal munch: find the largest pattern that matches a subtree of the IR tree, and replace it with the corresponding instruction.
- Dynamic programming algorithms to find the optimum instruction.

Instruction selection

Since instruction selection is highly target machine dependent, modern compilers implement domain-specific languages (DSL) to specify the patterns and their mapping to machine instructions.

Examples:

- LLVM has SelectionDAG phase and `tablegen`
- GCC has dedicated "Machine Description" files

Instruction Scheduling

Instruction Ordering

When a compiler emits the instructions corresponding to a program, it imposes a total order (of execution) on them.

It is *total* because once generated, one instruction is executed either before or after another instruction.

Instruction Ordering

When a compiler emits the instructions corresponding to a program, it imposes a total order (of execution) on them.

It is *total* because once generated, one instruction is executed either before or after another instruction.

However, that order is usually not the only valid one, in the sense that it can be changed without modifying the program's behavior.

Example: if two instructions i_1 and i_2 appear sequentially in that order and are independent, then it is possible to swap them.

Instruction Scheduling

Among all the valid permutations of the instructions composing a program - i.e. those that preserve the program's behavior - some can be more desirable than others.

Example: one order might lead to a faster program on some machine, because of architectural constraints.

Instruction Scheduling

Among all the valid permutations of the instructions composing a program - i.e. those that preserve the program's behavior - some can be more desirable than others.

Example: one order might lead to a faster program on some machine, because of architectural constraints.

Instruction scheduling: In what valid order should these instructions execute to optimize some metric?

Example: minimize pipeline stall.

Pipeline Stalls

Modern, pipelined CPU architectures to issue at least one instruction per clock cycle. This provides a form of **instruction-level parallelism**.

5-stage pipeline: fetch, decode, execute, memory access, and write back.

Pipeline Stalls

Modern, pipelined CPU architectures to issue at least one instruction per clock cycle. This provides a form of **instruction-level parallelism**.

5-stage pipeline: fetch, decode, execute, memory access, and write back.

However, there are many cases where the pipeline stalls

- Data dependency: an instruction can be executed only if the data it needs is ready.
- Memory access latency: data has to be fetched from memory, which can take several cycles.
- Instruction latency: some instructions, such as division, require several cycles to complete.

Scheduling Example

The following example illustrates how proper scheduling can reduce the time required to execute a piece of RTL code.

We assume the following delays for instructions:

Instruction kind	RTL notation	Delay
Memory load	$R_a \leftarrow Mem[R_b+c]$	3
Memory store	$Mem[R_b+c] \leftarrow R_a$	3
Multiplication	$R_a \leftarrow R_b \times R_c$	2
Addition	$R_a \leftarrow R_b + R_c$	1

Scheduling Example

Before scheduling:

Cycle	Instruction
1	`R1 <- Mem[RSP]`
4	`R1 <- R1 + R1`
5	`R2 <- Mem[RSP+1]`
8	`R1 <- R1 * R2`
9	`R2 <- Mem[RSP+2]`
12	`R1 <- R1 * R2`
13	`R2 <- Mem[RSP+3]`
16	`R1 <- R1 * R2`
18	`Mem[RSP+4] <- R1`

Scheduling Example

Before scheduling:

Cycle	Instruction
1	`R1 <- Mem[RSP]`
4	`R1 <- R1 + R1`
5	`R2 <- Mem[RSP+1]`
8	`R1 <- R1 * R2`
9	`R2 <- Mem[RSP+2]`
12	`R1 <- R1 * R2`
13	`R2 <- Mem[RSP+3]`
16	`R1 <- R1 * R2`
18	`Mem[RSP+4] <- R1`

After scheduling and renaming:

Cycle	Instruction
1	`R1 <- Mem[RSP]`
2	`R2 <- Mem[RSP+1]`
3	`R3 <- Mem[RSP+2]`
4	`R1 <- R1 + R1`
5	`R1 <- R1 * R2`
6	`R2 <- Mem[RSP+3]`
7	`R1 <- R1 * R3`
9	`R1 <- R1 * R2`
11	`Mem[RSP+4] <- R1`

Scheduling Example

Before scheduling:

Cycle	Instruction
1	`R1 <- Mem[RSP]`
4	`R1 <- R1 + R1`
5	`R2 <- Mem[RSP+1]`
8	`R1 <- R1 * R2`
9	`R2 <- Mem[RSP+2]`
12	`R1 <- R1 * R2`
13	`R2 <- Mem[RSP+3]`
16	`R1 <- R1 * R2`
18	`Mem[RSP+4] <- R1`

After scheduling and renaming:

Cycle	Instruction
1	`R1 <- Mem[RSP]`
2	`R2 <- Mem[RSP+1]`
3	`R3 <- Mem[RSP+2]`
4	`R1 <- R1 + R1`
5	`R1 <- R1 * R2`
6	`R2 <- Mem[RSP+3]`
7	`R1 <- R1 * R3`
9	`R1 <- R1 * R2`
11	`Mem[RSP+4] <- R1`

Result: issuing the program only takes 11 instead of 18 cycles!

Instruction Dependency

An instruction i_2 **depends** on an instruction i_1 when it is not possible to execute i_2 before i_1 without changing the behavior of the program.

Instruction Dependency

An instruction i_2 **depends** on an instruction i_1 when it is not possible to execute i_2 before i_1 without changing the behavior of the program.

The most common reason for dependency is **data dependency**: i_2 uses a value that is computed by i_1.

However, as we will see, there are other kinds of dependency.

Data Dependency

We distinguish three kinds of dependency from instruction i_2 to i_1:

- true dependency - i_2 reads a value written by i_1 (read after write or RAW),

```
R1 <- Mem[RSP]
R1 <- R1 + R1
```

Data Dependency

We distinguish three kinds of dependency from instruction i_2 to i_1:

- true dependency - i_2 reads a value written by i_1 (read after write or RAW),

```
R1 <- Mem[RSP]
R1 <- R1 + R1
```

- anti-dependency - i_2 writes a value read by i_1 (write after read or WAR),

```
R1 <- R1 + R1
R1 <- Mem[RSP+1]
```

Data Dependency

We distinguish three kinds of dependency from instruction i_2 to i_1:

- true dependency - i_2 reads a value written by i_1 (read after write or RAW),

```
R1 <- Mem[RSP]
R1 <- R1 + R1
```

- anti-dependency - i_2 writes a value read by i_1 (write after read or WAR),

```
R1 <- R1 + R1
R1 <- Mem[RSP+1]
```

- anti-dependency - i_2 writes a value written by i_1 (write after write or WAW).

```
R1 <- Mem[RSP]
R1 <- Mem[RSP+1]
```

Data Dependency

Impact on scheduling if i_2 depends on i_1:

- True dependency: i_1 must be scheduled before i_2.
- Anti-dependency: i_1 should not be scheduled after i_2 (but might not be scheduled).

Anti-dependency

Anti-dependency is not real dependency, in the sense that they do not arise from the flow of data. They are due to a single location being used to store different values.

Most of the time, anti-dependency can be removed by renaming locations - e.g. registers.

Anti-dependency

Anti-dependency is not real dependency, in the sense that they do not arise from the flow of data. They are due to a single location being used to store different values.

Most of the time, anti-dependency can be removed by renaming locations - e.g. registers.

In the example below, the program on the left contains a WAW anti-dependency between the two memory load instructions, that can be removed by renaming the second use of R_1.

```
R1 <- Mem[RSP]
R4 <- R4 + R1
R1 <- Mem[RSP+1]
R4 <- R4 + R1
```

```
R1 <- Mem[RSP]
R4 <- R4 + R1
R2 <- Mem[RSP+1] // rename R1 to R2
R4 <- R4 + R2    // rename R1 to R2
```

Computing Dependency

Identifying dependency among instructions that only access registers is easy.

Instructions that access memory are harder to handle. In general, it is not possible to know whether two such instructions refer to the same memory location.

Conservative approximations (alias analysis, not examined here) therefore have to be used.

Dependence Graph

The **dependence graph** is a directed graph representing dependency among instructions.

Its nodes are the instructions to schedule, and there is an edge from node n_1 to node n_2 iff the instruction of n_2 depends on n_1.

Any topological sort of the nodes of this graph represents a valid way to schedule the instructions.

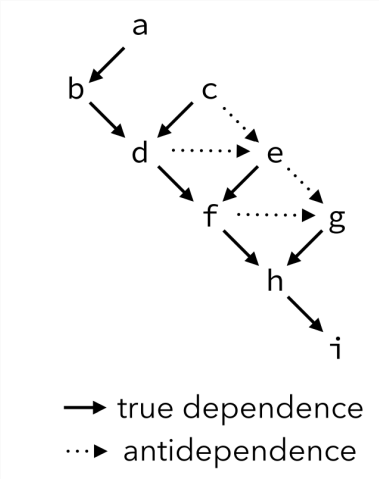
Dependence Graph Example

Name	Instruction
a	`R1 <- Mem[RSP]`
b	`R1 <- R1 + R1`
c	`R2 <- Mem[RSP+1]`
d	`R1 <- R1 * R2`
e	`R2 <- Mem[RSP+2]`
f	`R1 <- R1 * R2`
g	`R2 <- Mem[RSP+3]`
h	`R1 <- R1 * R2`
i	`Mem[RSP+4] <- R1`

True dep: read-after-write; Anti dep: write-after-read, write-after-write

Dependence Graph Example

Name	Instruction
a	`R1 <- Mem[RSP]`
b	`R1 <- R1 + R1`
c	`R2 <- Mem[RSP+1]`
d	`R1 <- R1 * R2`
e	`R2 <- Mem[RSP+2]`
f	`R1 <- R1 * R2`
g	`R2 <- Mem[RSP+3]`
h	`R1 <- R1 * R2`
i	`Mem[RSP+4] <- R1`



True dep: read-after-write; Anti dep: write-after-read, write-after-write

Difficulty Of Scheduling

Instruction scheduling: find a topological sort of the dependency graph that minimize some metric of cost.

Optimal instruction scheduling is **NP-complete**.

As always, this implies that we will use techniques based on heuristics to find a good - but sometimes not optimal - solution to that problem.

List Scheduling Algorithm

List scheduling is a greedy algorithm to schedule the instructions of a single basic block (so we don't have to worry about control flow).

- Its basic idea is to simulate the execution of the instructions, and to try to schedule instructions only when all their operands can be used without stalling the pipeline.
- Need to model hardware constraints, such as the latency of instructions, and the number of instructions that can be issued at each cycle.

List Scheduling Algorithm

The list scheduling algorithm maintains two lists:

- **ready** is the list of instructions that could be scheduled without stall, ordered by priority,
- **active** is the list of instructions that are being executed.

At each step, the highest-priority instruction from **ready** is scheduled, and moved to **active**, where it stays for a time equal to its delay.

Before scheduling is performed, renaming is done to remove all anti-dependency that can be removed.

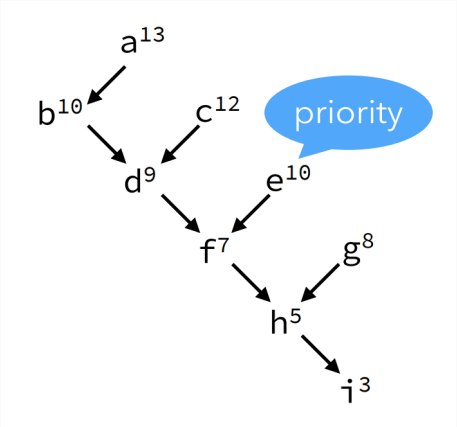
Prioritizing Instructions

Nodes (i.e. instructions) are sorted by priority in the **ready** list. Several schemes exist to compute the priority of a node, which can be equal to:

- the length of the longest latency-weighted path from it to a root of the dependence graph,
- the number of its immediate successors,
- the number of its descendants,
- its latency,
- etc.

Unfortunately, no single scheme is better for all cases.

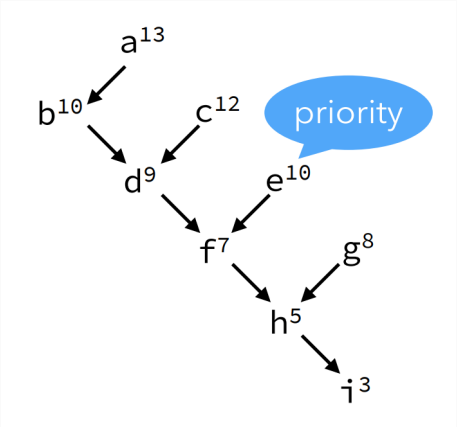
List Scheduling Example



Cycle	ready	active

A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

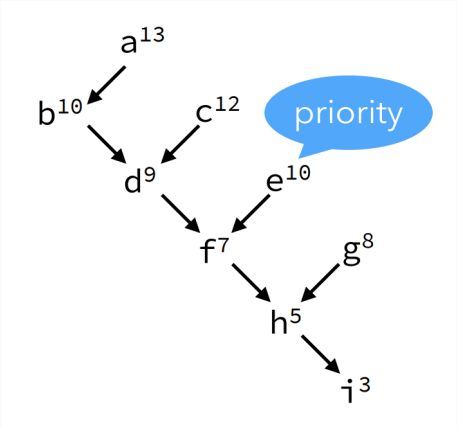
List Scheduling Example



Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]

A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

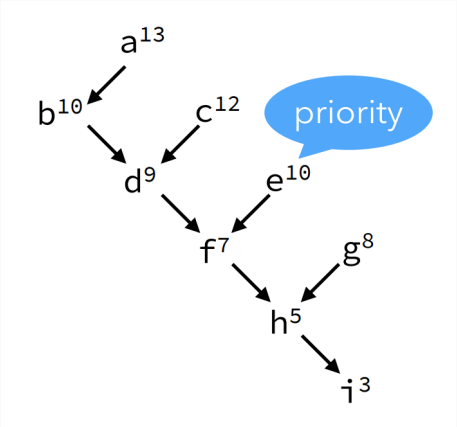
List Scheduling Example



Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]

A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

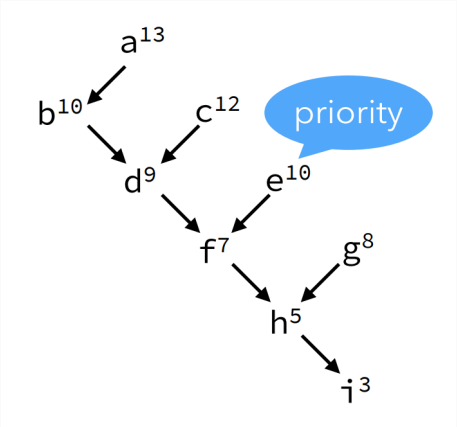
List Scheduling Example



Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]

A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

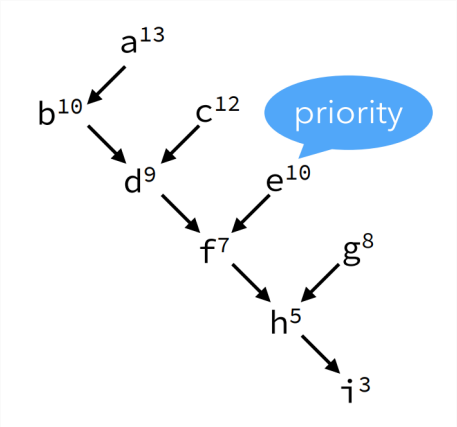
List Scheduling Example



Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]

A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

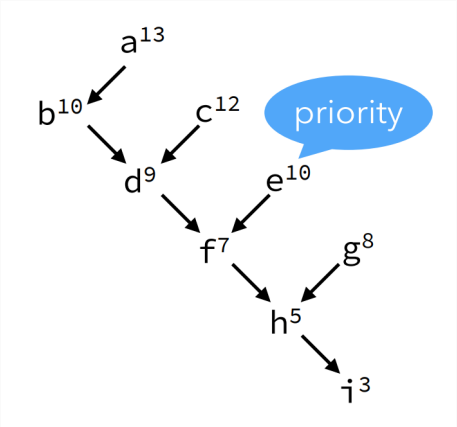
List Scheduling Example



Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]
5	[d^{9},g^{8}]	[d,e]

A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

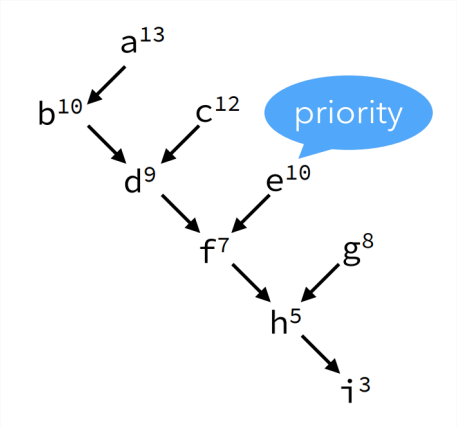
List Scheduling Example



Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]
5	[d^{9},g^{8}]	[d,e]
6	[g^{8}]	[d,g]

A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

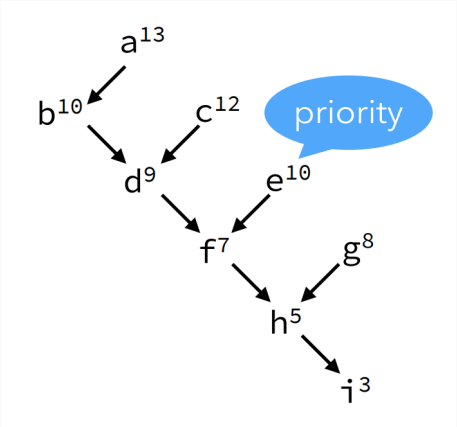
List Scheduling Example



Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]
5	[d^{9},g^{8}]	[d,e]
6	[g^{8}]	[d,g]
7	[f^{7}]	[f,g]

A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

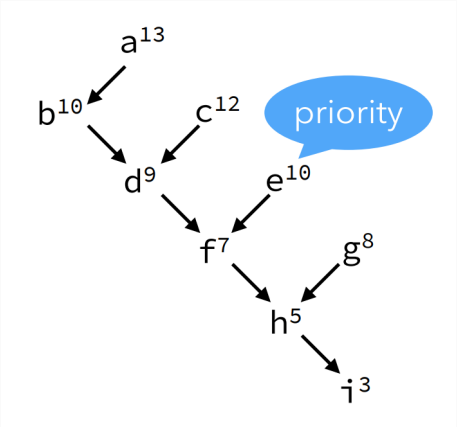
List Scheduling Example



Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]
5	[d^{9},g^{8}]	[d,e]
6	[g^{8}]	[d,g]
7	[f^{7}]	[f,g]
8	[]	[f,g]

A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

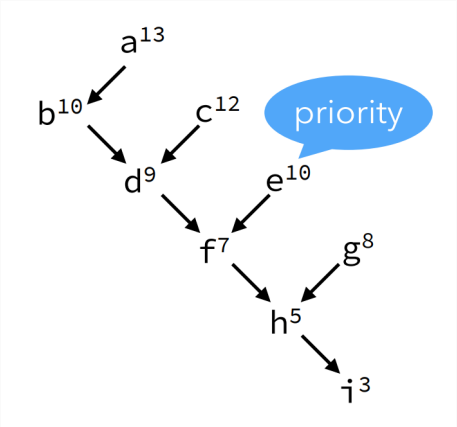
List Scheduling Example



Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]
5	[d^{9},g^{8}]	[d,e]
6	[g^{8}]	[d,g]
7	[f^{7}]	[f,g]
8	[]	[f,g]
9	[h^{5}]	[h]

A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

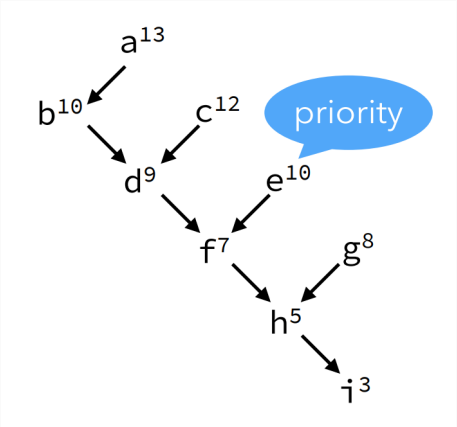
List Scheduling Example



A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]
5	[d^{9},g^{8}]	[d,e]
6	[g^{8}]	[d,g]
7	[f^{7}]	[f,g]
8	[]	[f,g]
9	[h^{5}]	[h]
10	[]	[h]

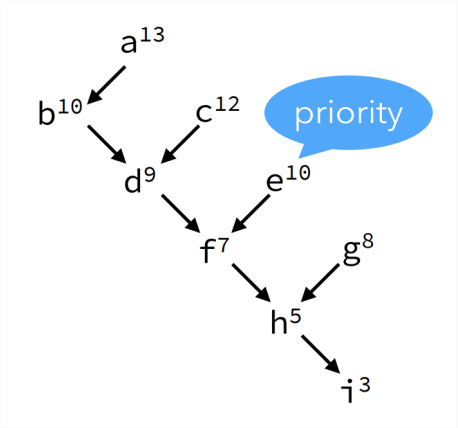
List Scheduling Example



A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]
5	[d^{9},g^{8}]	[d,e]
6	[g^{8}]	[d,g]
7	[f^{7}]	[f,g]
8	[]	[f,g]
9	[h^{5}]	[h]
10	[]	[h]
11	[i^{3}]	[i]

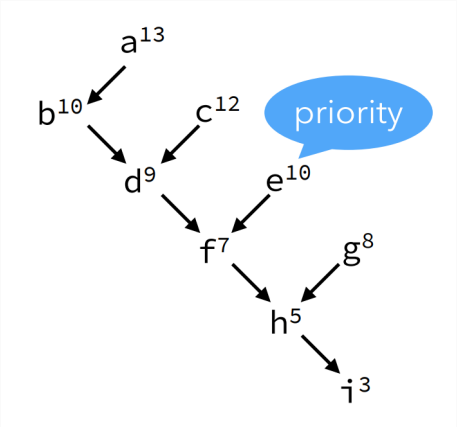
List Scheduling Example



A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]
5	[d^{9},g^{8}]	[d,e]
6	[g^{8}]	[d,g]
7	[f^{7}]	[f,g]
8	[]	[f,g]
9	[h^{5}]	[h]
10	[]	[h]
11	[i^{3}]	[i]
12	[]	[i]

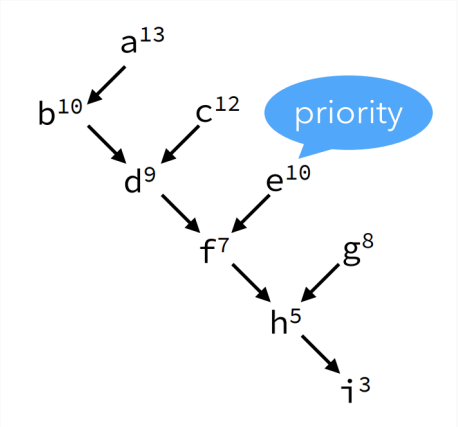
List Scheduling Example



A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]
5	[d^{9},g^{8}]	[d,e]
6	[g^{8}]	[d,g]
7	[f^{7}]	[f,g]
8	[]	[f,g]
9	[h^{5}]	[h]
10	[]	[h]
11	[i^{3}]	[i]
12	[]	[i]
13	[]	[i]

List Scheduling Example



A node's priority is the length of the longest latency weighted path from it to a root of the dependence graph

Cycle	ready	active
1	[a^{13},c^{12},e^{10},g^{8}]	[a]
2	[c^{12},e^{10},g^{8}]	[a,c]
3	[e^{10},g^{8}]	[a,c,e]
4	[b^{10},g^{8}]	[b,c,e]
5	[d^{9},g^{8}]	[d,e]
6	[g^{8}]	[d,g]
7	[f^{7}]	[f,g]
8	[]	[f,g]
9	[h^{5}]	[h]
10	[]	[h]
11	[i^{3}]	[i]
12	[]	[i]
13	[]	[i]

Scheduling conflicts

It is hard to decide whether scheduling should be done before or after register allocation.

If register allocation is done first, it can introduce anti-dependency when reusing registers.

If scheduling is done first, register allocation can introduce spilling code, destroying the schedule.

Solution: schedule first, then allocate registers and schedule once more if spilling was necessary.

Next week

- What we didn't talk about in this class and research topics in compilers
- Final exam review