CENG513 Compiler Design and Construction
Instruction Scheduling

Note by Işıl ÖZ:
Our slides are adapted from Cooper and Torczon's slides that are prepared for COMP 412 at Rice.

What Makes Code Run Fast?

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
- Execution time is *order-dependent* *(and has been since the 60's)*

Operation	Cycles
load	3
store	3
loadI	1
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

- Loads & stores may or may not block on issue
  - > Non-blocking ⇒ fill those issue slots
- Branch costs vary with path taken
- Branches typically have delay slots
  - > Fill slots with unrelated operations
  - > Percolates branch upward
- Scheduler should hide the latencies

Example

$$w \leftarrow w * 2 * x * y * z$$

Simple schedule

Start	Original Code
1	loadAI r_{arp}, @w ⇒ r_1
4	add r_1, r_1 ⇒ r_1
5	loadAI r_{arp}, @x ⇒ r_2
8	mult r_1, r_2 ⇒ r_1
9	loadAI r_{arp}, @y ⇒ r_2
12	mult r_1, r_2 ⇒ r_1
13	loadAI r_{arp}, @z ⇒ r_2
16	mult r_1, r_2 ⇒ r_1
18	storeAI r_1 ⇒ r_{arp},0

2 registers, 20 cycles

Schedule loads early

Start	Scheduled Code
1	loadAI r_{arp}, @w ⇒ r_1
2	loadAI r_{arp}, @x ⇒ r_2
3	loadAI r_{arp}, @y ⇒ r_3
4	add r_1, r_1 ⇒ r_1
5	mult r_1, r_2 ⇒ r_1
6	loadAI r_{arp}, @z ⇒ r_2
7	mult r_1, r_3 ⇒ r_1
9	mult r_1, r_2 ⇒ r_1
11	storeAI r_1 ⇒ r_{arp},0

3 registers, 13 cycles

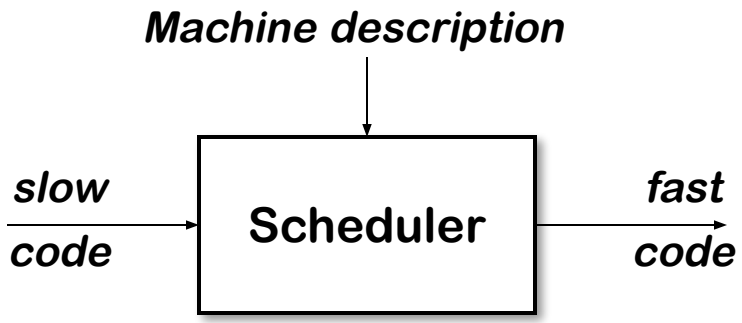
Reordering operations for speed is called *instruction scheduling*

Instruction Scheduling (Engineer's View)

The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time

The Concept



The Task

- Produce correct code
- Minimize wasted cycles
- Avoid spilling registers
- Operate efficiently

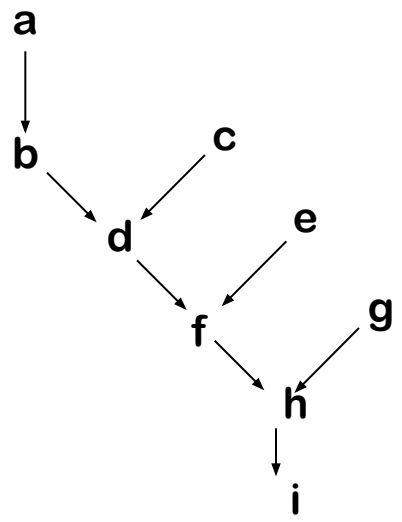
Instruction Scheduling (Abstract View)

To capture properties of the code, build a precedence graph *G*

- Nodes $n \in G$ are operations with *type(n)*: *a functional unit of type* and *delay(n)*: *cycles to complete*
- An edge $e = (n_1, n_2) \in G$ if & only if n_2 uses the result of n_1

a:	loadAI	r_0, 0	⇒	r_1
b:	add	r_1, r_1	⇒	r_1
c:	loadAI	r_0, 8	⇒	r_2
d:	mult	r_1, r_2	⇒	r_1
e:	loadAI	r_0, 16	⇒	r_2
f:	mult	r_1, r_2	⇒	r_1
g:	loadAI	r_0, 24	⇒	r_2
h:	mult	r_1, r_2	⇒	r_1
i:	storeAI	r_1	⇒	r_0,0

The Code



The Precedence Graph

Instruction Scheduling (Definitions)

A *correct schedule* S maps each $n \in N$ into a non-negative integer representing its cycle number in which it should be issued

1. $S(n) \geq 1$, for all $n \in N$, forbids operations from being issued before execution starts
2. If $(n_1, n_2) \in E$, $S(n_1) + delay(n_1) \leq S(n_2)$
3. For each type t, there are no more operations of type t in any cycle than the target machine can issue

The *length* of a schedule S, denoted $L(S)$, is

$$L(S) = max_{n \in N} (S(n) + delay(n))$$

(assuming the first instruction issues in cycle one)

Instruction Scheduling (Definitions)

The goal is to find the shortest possible correct schedule

Improve instruction-level parallelism, avoid pipeline stalls

S is *time-optimal* if $L(S) \leq L(S_1)$, for all other schedules S_1

A schedule might also be optimal in terms of registers, power, or space….

Instruction Scheduling (What's so difficult?)

Critical Points

- All operands must be available
- Multiple operations can be *ready*
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- Operands can have multiple predecessors

Together, these issues make scheduling *hard* (NP-Complete)

Local scheduling is the simple case

- Restricted to straight-line code (a single basic block)
- Consistent and predictable latencies

Instruction Scheduling: The Big Picture

1. Build a precedence graph, *P*

2. Compute a *priority function* over the nodes in *P*

3. Use list scheduling to construct a schedule, 1 cycle at a time
   a. Use a queue of operations that are ready (*ready list*)
   b. At each cycle
      I. Choose the highest priority ready operation & schedule it
      II. Update the ready queue

Local list scheduling

- The dominant algorithm for years
- A greedy, heuristic, local technique

List Scheduling Algorithm

```
Cycle ← 1
Ready ← leaves of P
Active ← Ø

while (Ready ∪ Active ≠ Ø)
    if (Ready ≠ Ø) then
       remove an op from Ready
       S(op) ← Cycle
       Active ¬ Active  ∪ op

    Cycle ← Cycle + 1

    for each op ∈ Active
        if (S(op) + delay(op) ≤ Cycle) then
           remove op from Active
           for each successor s of op in P
               if (s is ready) then
                  Ready ← Ready ∪ s
```

Removal in priority order

op has completed execution

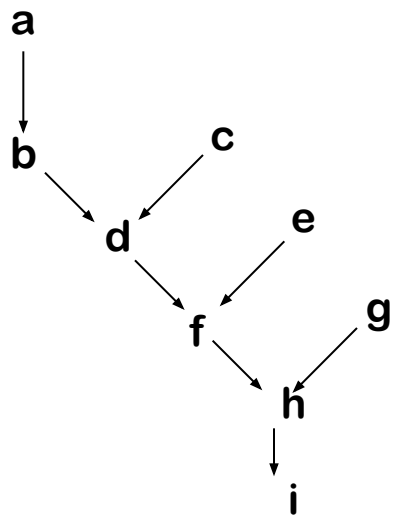
If successor's operands are "ready", add it to Ready

Scheduling Example

1. Build the precedence graph

a:	loadAI	r_0, 0	⇒	r_1
b:	add	r_1, r_1	⇒	r_1
c:	loadAI	r_0, 8	⇒	r_2
d:	mult	r_1, r_2	⇒	r_1
e:	loadAI	r_0, 16	⇒	r_2
f:	mult	r_1, r_2	⇒	r_1
g:	loadAI	r_0, 24	⇒	r_2
h:	mult	r_1, r_2	⇒	r_1
i:	storeAI	r_1	⇒	r_0,0

The Code



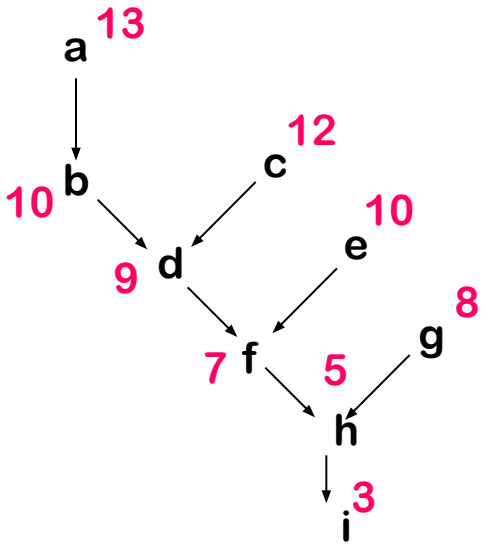
The Precedence Graph

Scheduling Example

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0, 0   ⇒ r1
b:  add      r1, r1  ⇒ r1
c:  loadAI   r0, 8   ⇒ r2
d:  mult     r1, r2  ⇒ r1
e:  loadAI   r0, 16  ⇒ r2
f:  mult     r1, r2  ⇒ r1
g:  loadAI   r0, 24  ⇒ r2
h:  mult     r1, r2  ⇒ r1
i:  storeAI  r1      ⇒ r0,0
```

The Code



The Precedence Graph

Scheduling Example

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling

a:	loadAI	r_0, 0	⇒	r_1
b:	add	r_1, r_1	⇒	r_1
c:	loadAI	r_0, 8	⇒	r_2
d:	mult	r_1, r_2	⇒	r_1
e:	loadAI	r_0, 16	⇒	r_2
f:	mult	r_1, r_2	⇒	r_1
g:	loadAI	r_0, 24	⇒	r_2
h:	mult	r_1, r_2	⇒	r_1
i:	storeAI	r_1	⇒	r_0,0

Start	Scheduled Code			
1	loadAI	r_{arp}, @w	⇒	r_1
2	loadAI	r_{arp}, @x	⇒	r_2
3	loadAI	r_{arp}, @y	⇒	r_3
4	add	r_1, r_1	⇒	r_1
5	mult	r_1, r_2	⇒	r_1
6	loadAI	r_{arp}, @z	⇒	r_2
7	mult	r_1, r_3	⇒	r_1
9	mult	r_1, r_2	⇒	r_1
11	storeAI	r_1	⇒	r_{arp},0

More List Scheduling

List scheduling breaks down into two distinct classes

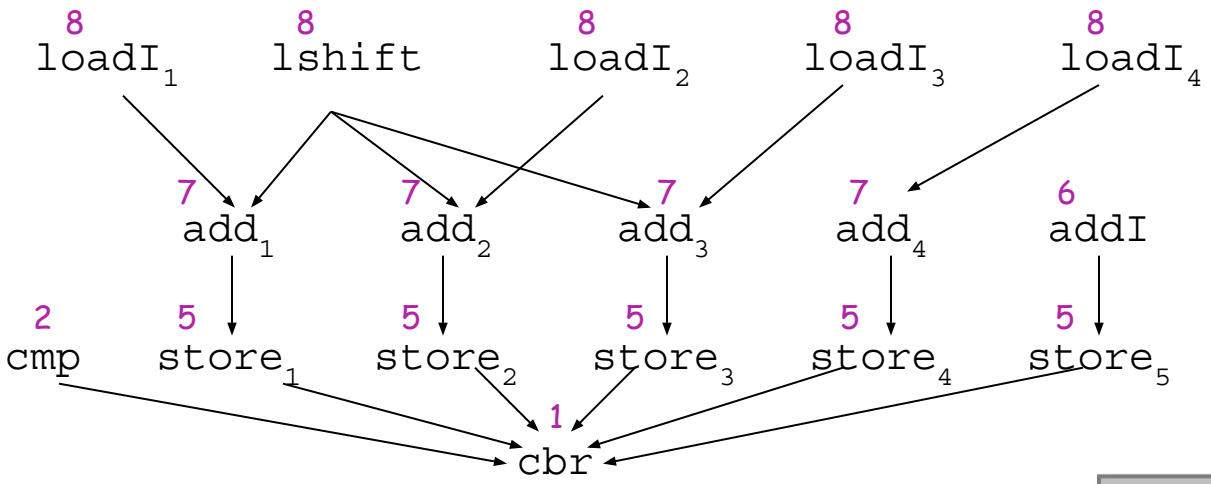
Forward list scheduling	Backward list scheduling
• Start with available operations	• Start with no successors
• Work forward in time	• Work backward in time
• Ready ⇒ all operands available	• Ready ⇒ latency covers uses

Variations on list scheduling

- Prioritize critical path(s)
- Schedule last use as soon as possible
- Depth first in precedence graph (minimize registers)
- Breadth first in precedence graph (minimize interlocks)
- Prefer operation with most successors

Local Scheduling

Forward and backward can produce different results



Latency to the cbr

Subscript to identify

Block from SPEC benchmark "go"

Operation	load	loadI	add	addI	store	cmp
Latency	1	1	2	1	4	1

Local Scheduling

Forward Schedule

	Int	Int	Mem
1	$loadI_1$	lshift	
2	$loadI_2$	$loadI_3$	
3	$loadI_4$	add_1	
4	add_2	add_3	
5	add_4	addI	$store_1$
6	cmp		$store_2$
7			$store_3$
8			$store_4$
9			$store_5$
10			
11			
12			
13	cbr		

Backward Schedule

	Int	Int	Mem
1	$loadI_4$		
2	addI	lshift	
3	add_4	$loadI_3$	
4	add_3	$loadI_2$	$store_5$
5	add_2	$loadI_1$	$store_4$
6	add_1		$store_3$
7			$store_2$
8			$store_1$
9			
10			
11	cmp		
12	cbr		

Using "latency to root" as the priority function

List Scheduling Variant

Schielke's RBF Algorithm

- Recognize that forward & backward both have their place
- Recognize that tie-breaking matters & that, while we can *rationalize* various tie-breakers, we do not understand them
  - Part of approximating the solution to an NP-complete problem

The Algorithm

- Run forward scheduler 5 times, breaking ties randomly
- Run backward scheduler 5 time, breaking ties randomly
- Keep the best schedule

RBF means "randomized backward & forward"

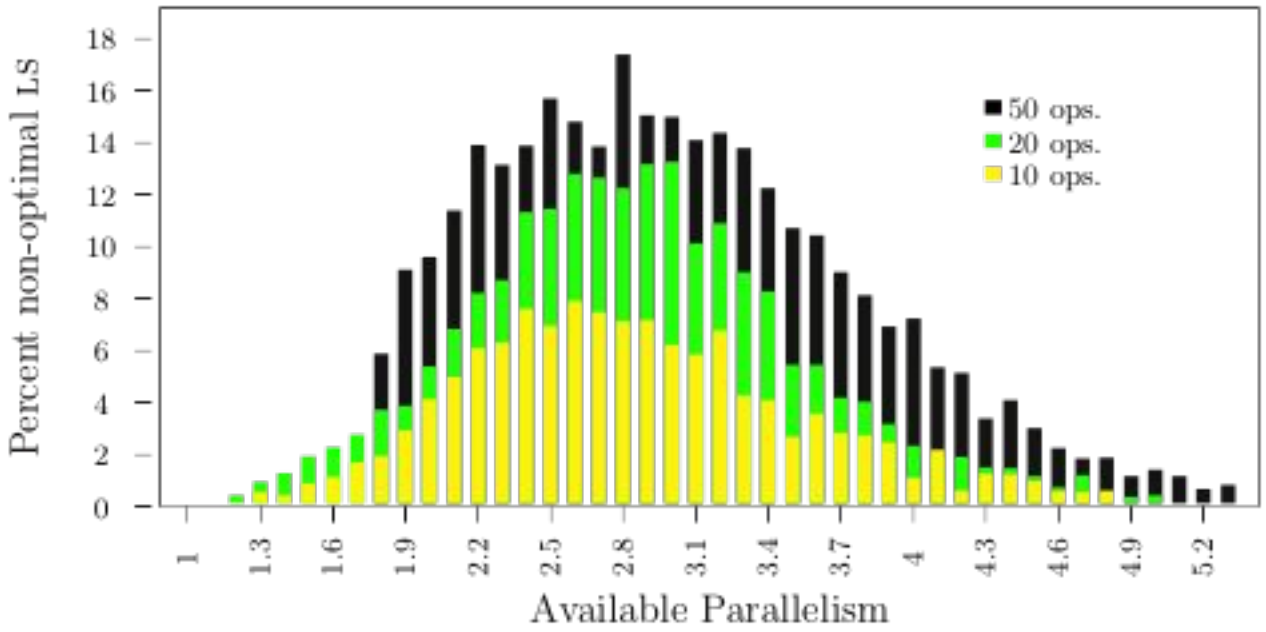
Randomized
Backward &
Forward

The implementation must be scrupulous about randomizing *all* choices.

How Good is List Scheduling? Schielke's Study

Non-optimal list schedules (%) versus available parallelism

1 functional unit, randomly generated blocks of 10, 20, 50 ops



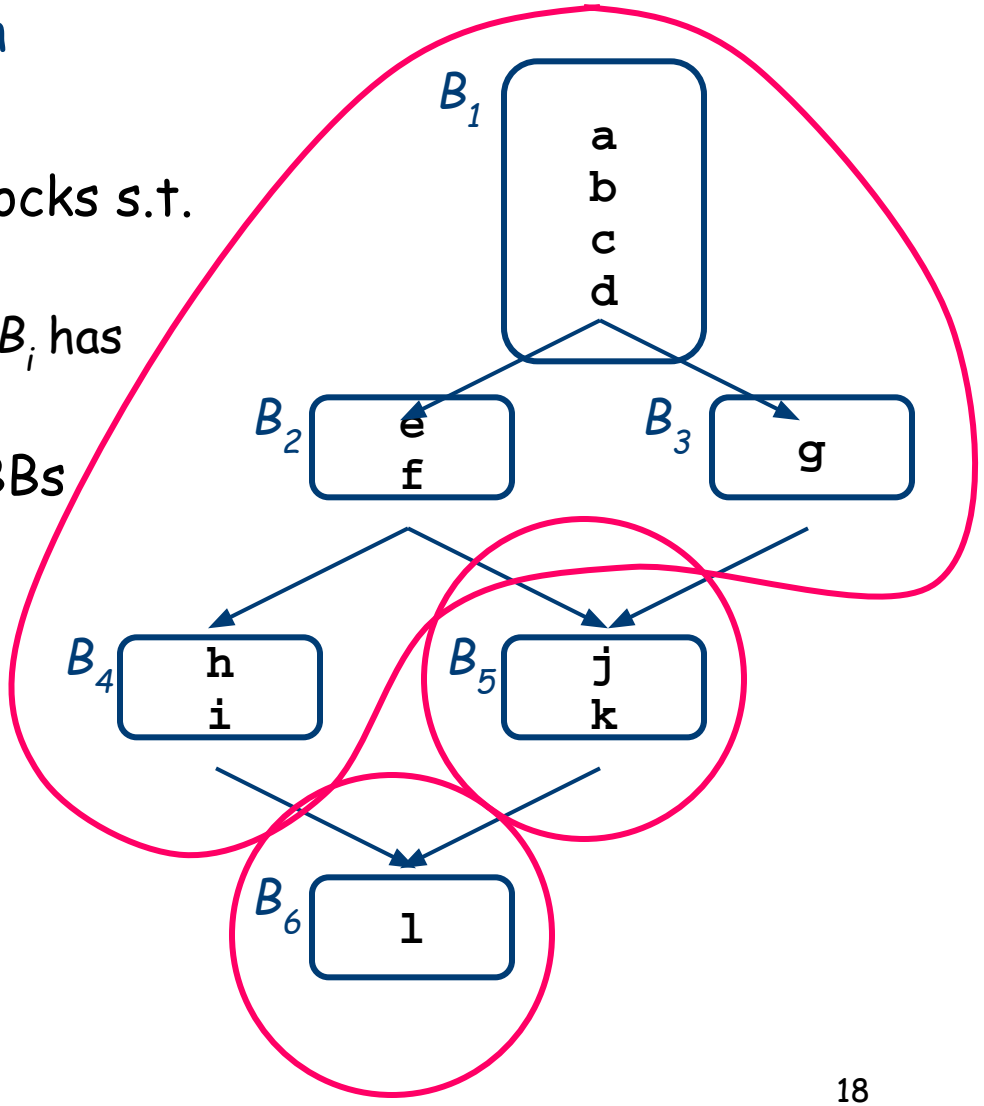
- 85,000 randomly generated blocks
- RBF found optimal schedules for > 80%

Tie-breaking matters because it affects the choice when queue has > 1 element

Scheduling Larger Regions

One step beyond a block is an Extended Basic Block (EBB)

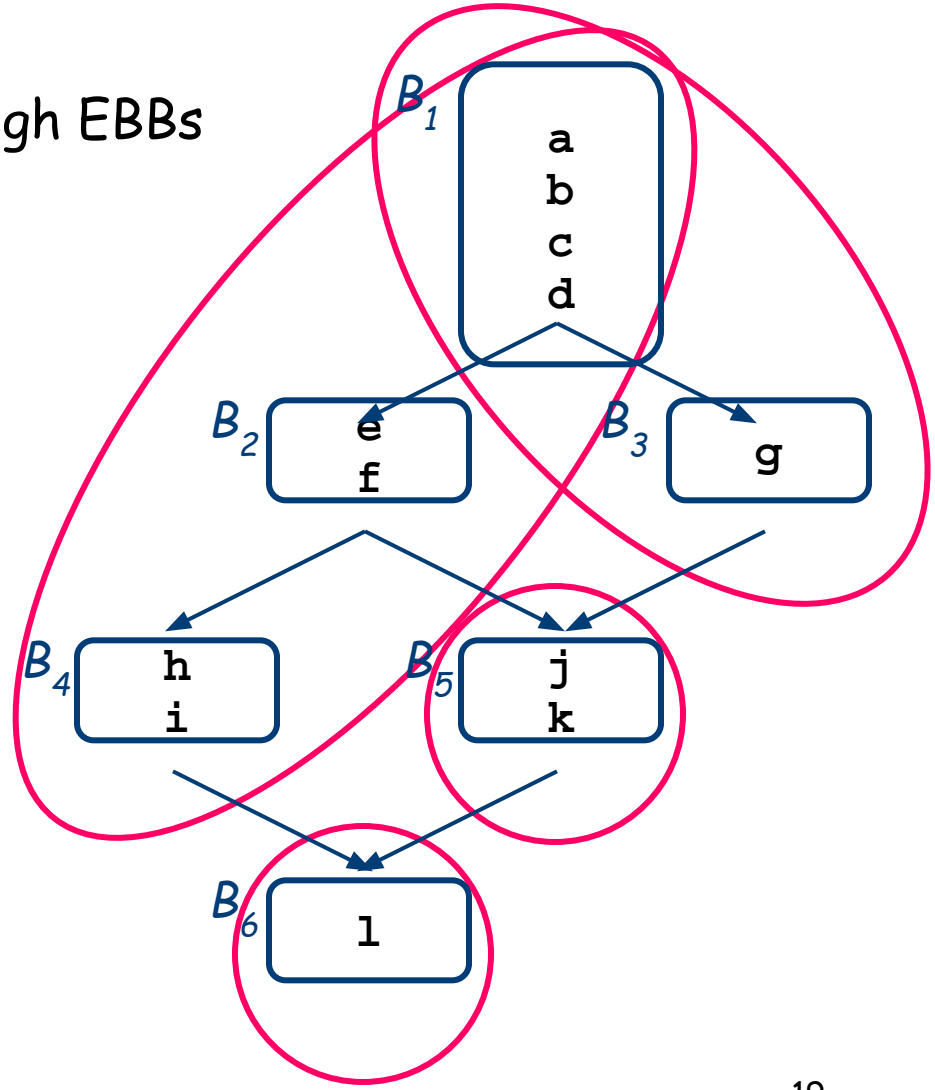
- EBB is a maximal set of blocks s.t.
  - Set has a single entry, B_i
  - Each block B_j other than B_i has exactly one predecessor
- Example CFG has three EBBs



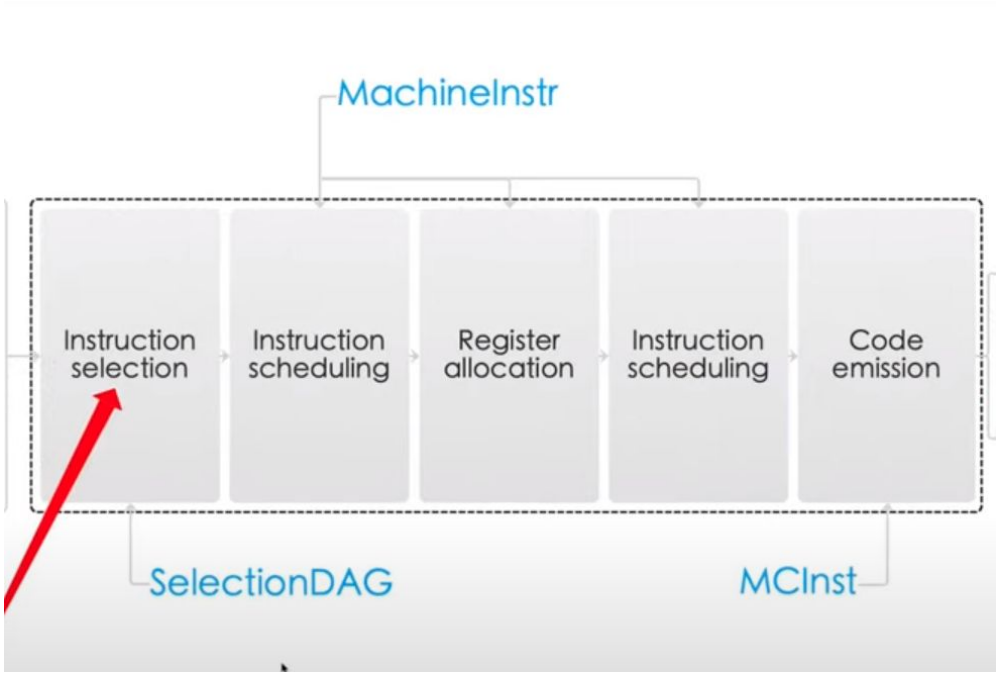
Scheduling Larger Regions

Superlocal Scheduling

- Schedule entire paths through EBBs
- Example has four EBB paths



LLVM Instruction Selection and Scheduling

- 

SelectionDAG Select Phase

- LLVM IR

```
%t1 = fadd float %W, %X
%t2 = fmul float %t1, %Y
%t3 = fadd float %t2, %Z
```

- SelectionDAG

```
(fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z)
```

- MachineInstr

```
(FMADDS (FADDS W, X), Y, Z)
```

- PowerPC instruction definitions

```
def FMADDS : AForm_1<59, 29,
                     (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRC, F4RC:$FRB),
                     "fmadds $FRT, $FRA, $FRC, $FRB",
                     [(set F4RC:$FRT, (fadd (fmul F4RC:$FRA, F4RC:$FRC),
                                            F4RC:$FRB))]>;
def FADDS : AForm_2<59, 21,
                     (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRB),
                     "fadds $FRT, $FRA, $FRB",
                     [(set F4RC:$FRT, (fadd F4RC:$FRA, F4RC:$FRB))]>;
```

SelectionDAG Instruction Selection Process

- Build initial DAG - performs a simple translation from the input LLVM code to an illegal SelectionDAG
- Optimize SelectionDAG - performs simple optimizations on the SelectionDAG to simplify it.
- Legalize SelectionDAG Types - transforms SelectionDAG nodes to eliminate any types that are unsupported on the target.
- Optimize SelectionDAG - The SelectionDAG optimizer is run to clean up redundancies exposed by type legalization.
- Legalize SelectionDAG Ops - transforms SelectionDAG nodes to eliminate any operations that are unsupported on the target.
- Optimize SelectionDAG - The SelectionDAG optimizer is run to eliminate inefficiencies introduced by operation legalization.

SelectionDAG Instruction Selection Process

- Select instructions from DAG - Finally, the target instruction selector matches the DAG operations to target instructions. This process translates the target-independent input DAG into another DAG of target instructions.
- SelectionDAG Scheduling and Formation - The last phase assigns a linear order to the instructions in the target-instruction DAG and emits them into the MachineFunction being compiled.

References

Chapter sections from the book:

- 12.1-12.3

Selected videos from compiler course from California State University:

- https://www.youtube.com/watch?v=0wYgorCBaxM&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=33
- https://www.youtube.com/watch?v=IezEeP2R4iA&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=34

LLVM instruction selection and scheduling

- https://llvm.org/docs/CodeGenerator.html#instruction-selection