Compiler Construction

Chapter 12: Instruction scheduling

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

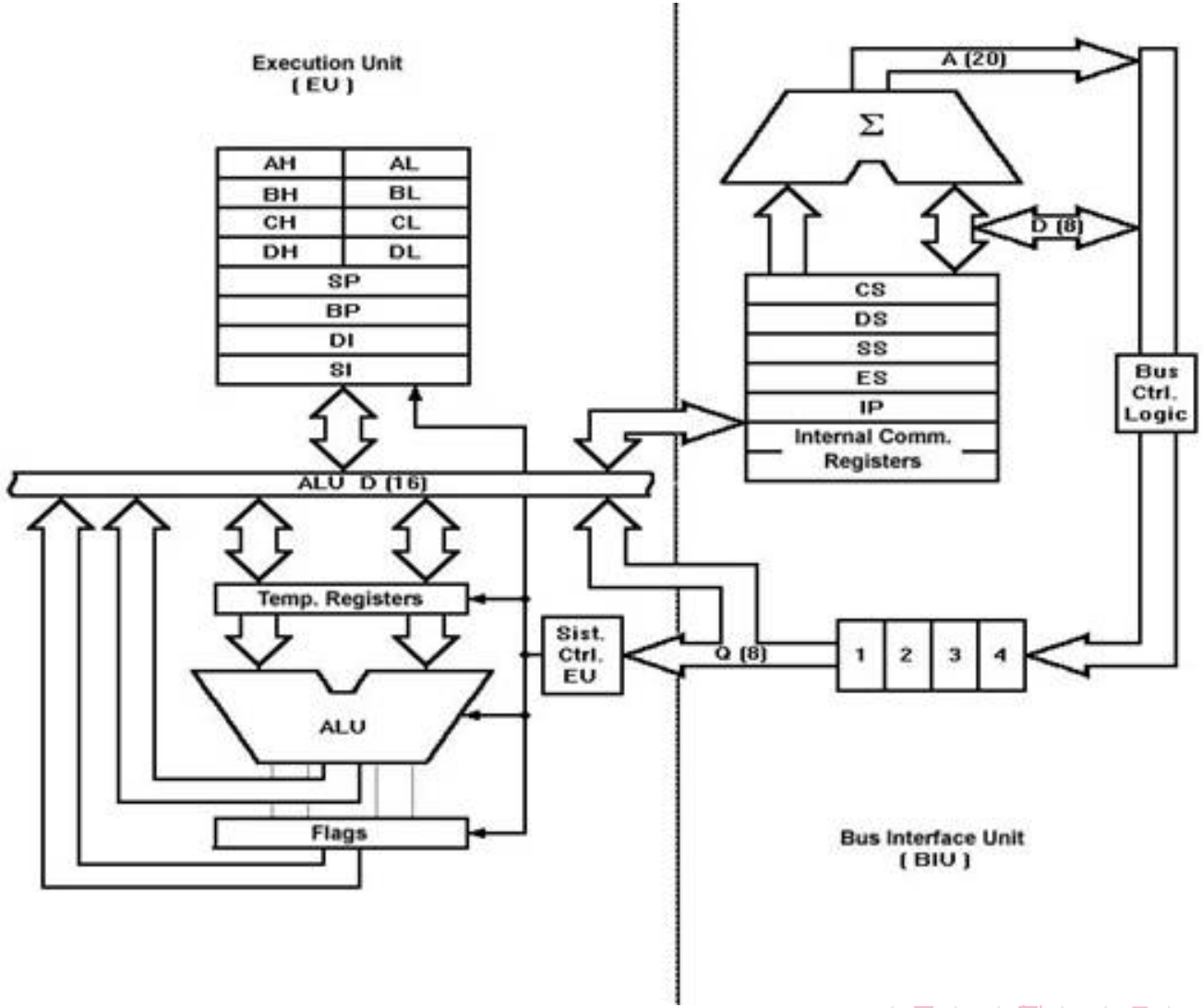
Second semester, 2022

Introduction

- Instruction reordering
- Operands must be ready before performing an operation
  - A processor may stall the premature operation
  - Explicit NOP (null operation)
  - Reorder instructions to avoid stall/nop
- Major algorithm: List scheduling
  - Input: partially ordered list of instructions
  - Output: ordered list of instructions

Architectures that affect performance

Pipelined execution



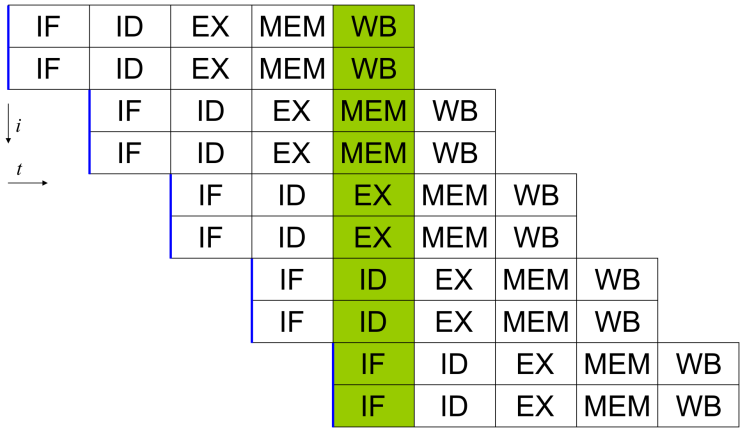
Architectures that affect performance

Memory operation

- Cache hit/missed
- Average latency estimation

Multiple functional units

- RISC
- Superscalar processor
- Very-long-instruction-word (VLIW) processor



By Amit6, original version (File:Superscalarpipeline.png) by User:Poil - Own work, CC BY-SA 3.0,

`https://commons.wikimedia.org/w/index.php?curid=9748747`

Example

- The processor has a single functional unit
- Loads and stores take three cycles
- A multiply takes two cycles
- Other operations take a cycle

Start	Operations			
1	loadAI	r_{arp}, @a	⇒	r_1
4	add	r_1, r_1	⇒	r_1
5	loadAI	r_{arp}, @b	⇒	r_2
8	mult	r_1, r_2	⇒	r_1
10	loadAI	r_{arp}, @c	⇒	r_2
13	mult	r_1, r_2	⇒	r_1
15	loadAI	r_{arp}, @d	⇒	r_2
18	mult	r_1, r_2	⇒	r_1
20	storeAI	r_1	⇒	r_{arp}, @a

(a) Original Code

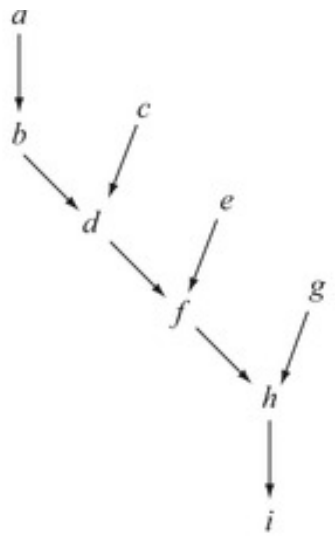
Start	Operations			
1	loadAI	r_{arp}, @a	⇒	r_1
2	loadAI	r_{arp}, @b	⇒	r_2
3	loadAI	r_{arp}, @c	⇒	r_3
4	add	r_1, r_1	⇒	r_1
5	mult	r_1, r_2	⇒	r_1
6	loadAI	r_{arp}, @d	⇒	r_2
7	mult	r_1, r_3	⇒	r_1
9	mult	r_1, r_2	⇒	r_1
11	storeAI	r_1	⇒	r_{arp}, @a

(b) Scheduled Code

Scheduling problem

a:	loadAI	r_{arp}, @a	⇒	r_1
b:	add	r_1, r_1	⇒	r_1
c:	loadAI	r_{arp}, @b	⇒	r_2
d:	mult	r_1, r_2	⇒	r_1
e:	loadAI	r_{arp}, @c	⇒	r_3
f:	mult	r_1, r_2	⇒	r_1
g:	loadAI	r_{arp}, @d	⇒	r_2
h:	mult	r_1, r_2	⇒	r_1
i:	storeAI	r_1	⇒	r_{arp}, @a

(a) Example Code



(b) Its Dependence Graph

Given

- A dependence graph $\mathcal{D}$ whose directed edge (x, y) indicates that the operation y uses the value produced by x.
- Each node has two attributes, a type and a delay.
- For a node n, the operation corresponding to n must execute on a functional unit specified by $type(n)$
- The operation n requires $delay(n)$ cycle to complete.

Scheduling problem

A schedule S maps each node $n \in N$ to a non-negative integer that denotes the cycle in which it should be issued

1. $S(n) \geq 1$ and at least one operation n' with $S(n') = 1$
2. If $(n_1, n_2) \in E$ then $S(n_1) + delay(n_1) \leq S(n_2)$
3. Each instruction contains no more operations of each type t than the target machine can issue in a cycle

The schedule length is

$$L(S) = \max_{n \in N} \left(S(n) + delay(n) - 1 \right)$$

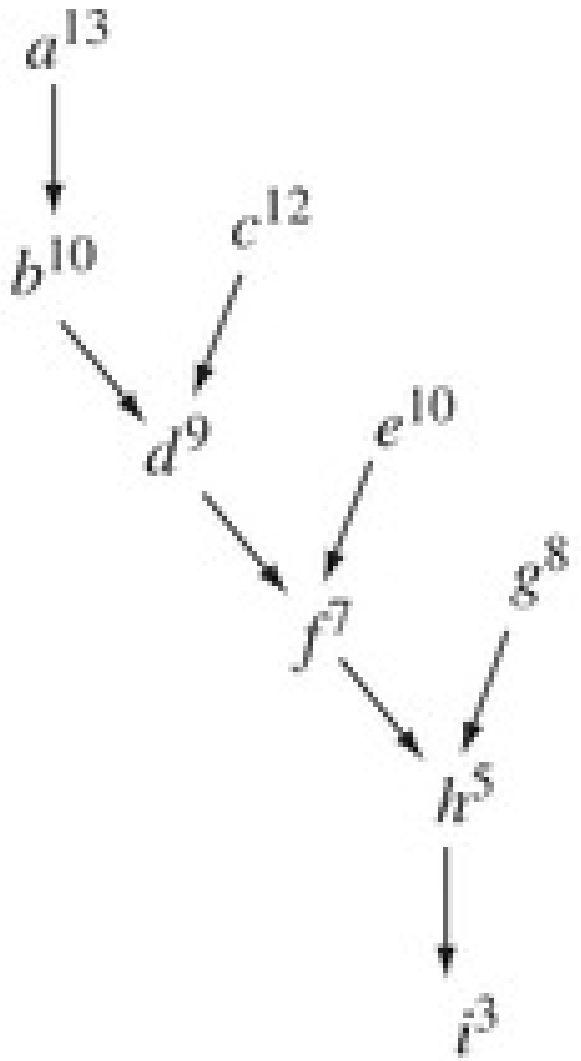
Local instruction scheduling is NP-complete.

Schedule quality

- Schedule length: time required to complete the task
- Demands for registers
- Energy consumption

Critical path

The longest path through the dependence graph



Local list scheduling

Start	Operations			
1	loadAI	r_{arp}, @a	⇒	r_1
4	add	r_1, r_1	⇒	r_1
5	loadAI	r_{arp}, @b	⇒	r_2
8	mult	r_1, r_2	⇒	r_1
10	loadAI	r_{arp}, @c	⇒	r_2
13	mult	r_1, r_2	⇒	r_1
15	loadAI	r_{arp}, @d	⇒	r_2
18	mult	r_1, r_2	⇒	r_1
20	storeAI	r_1	⇒	r_{arp}, @a

(a) Original Code

Start	Operations			
1	loadAI	r_{arp}, @a	⇒	r_1
2	loadAI	r_{arp}, @b	⇒	r_2
3	loadAI	r_{arp}, @c	⇒	r_3
4	add	r_1, r_1	⇒	r_1
5	mult	r_1, r_2	⇒	r_1
6	loadAI	r_{arp}, @d	⇒	r_2
7	mult	r_1, r_3	⇒	r_1
9	mult	r_1, r_2	⇒	r_1
11	storeAI	r_1	⇒	r_{arp}, @a

(b) Scheduled Code

Assume that

- load/store requires cycles
- add requires 1 cycle
- multiply requires 2 cycles

Local list scheduling

1. Rename to avoid antidependents
2. Build a dependence graph
3. Assign priorities
4. Iteratively select an operation and schedule

Renaming

```
VName ← 0
for i ← 0 to max source-register number do
    SToV[i] ← invalid                    // initialization
for each Op in the block, bottom to top do
    for each definition, O, in Op do     // do defs first
        if SToV[O] = invalid then        // invalid def indicates
            SToV[O] ← VName++            // an unused value
        O ← SToV[O]                      // O gets its new name
        SToV[O] ← invalid                // next ref is a new name
    for each use, O, in OP do            // do uses second
        if SToV[O] = invalid then        // start a new value
            SToV[O] ← VName++
        O ← SToV[O]                      // O gets its new name
```

■ **FIGURE 12.3** Renaming for List Scheduling.

Renaming

a:	loadAI	$r_{arp}, 4$	$\Rightarrow$	r_1
b:	add	r_1, r_1	$\Rightarrow$	r_1
c:	loadAI	$r_{arp}, 8$	$\Rightarrow$	r_2
d:	mult	r_1, r_2	$\Rightarrow$	r_1
e:	loadAI	$r_{arp}, 12$	$\Rightarrow$	r_2
f:	mult	r_1, r_2	$\Rightarrow$	r_1
g:	loadAI	$r_{arp}, 16$	$\Rightarrow$	r_2
h:	mult	r_1, r_2	$\Rightarrow$	r_1
i:	storeAI	r_1	$\Rightarrow$	$r_{arp}, 4$

(a) Example Code

loadAI	$r_{arp}, 4$	$\Rightarrow$	r_7
add	r_7, r_7	$\Rightarrow$	r_5
loadAI	$r_{arp}, 8$	$\Rightarrow$	r_6
mult	r_5, r_6	$\Rightarrow$	r_3
loadAI	$r_{arp}, 12$	$\Rightarrow$	r_4
mult	r_3, r_4	$\Rightarrow$	r_1
loadAI	$r_{arp}, 16$	$\Rightarrow$	r_2
mult	r_1, r_2	$\Rightarrow$	r_0
storeAI	r_0	$\Rightarrow$	$r_{arp}, 4$

Example After Renaming

Building the dependence graph

```
create an empty map, M                          // definitions to nodes
create a node, undef, in 𝒟                      // for an undefined value

for each operation O, top to bottom do          // walk the block
    create a node n for O, in 𝒟                 // n represents O
    for each name, d, defined in O do
        set M(d) to n
    for each name, u, used in O do              // true dependence
        if M(u) is undefined then
            set M(u) to undef
        add an edge (n,M(u)) to 𝒟
    if O is a memory operation then             // antidependences
        add serialization edges as needed
```

■ **FIGURE 12.4** Building the Dependence Graph After Renaming.

Computing priorities

- Tie breaking strategy
- Latency-weighted depth, number of descendants, breadth-first order, depth-first order

List scheduling algorithm

```
Cycle ← 1
Ready ← leaves of 𝒟
Active ← ∅

while (Ready ∪ Active ≠ ∅) do
    for each functional unit, f, do
        if there is an op in Ready for f then
            let O be the highest priority op      // choose O by priority
              in Ready that can execute on f
            remove O from Ready                   // schedule O in Cycle
            S(O) ← Cycle
            Active ← Active ∪ {O}
    Cycle ← Cycle + 1                             // start next Cycle
    for each O ∈ Active do
        if S(O) + delay(O) ≤ Cycle then           // update Ready list
            remove O from Active
            for each successor s of O in 𝒟 do
                if s is ready
                    then add s to Ready
```

■ **FIGURE 12.6** The List-Scheduling Algorithm.

Approximation of delays

Load operations

```
for each load operation, l, in the block do
    delay(l) ← 1

for each operation i in 𝒟 do
    let 𝒟_i be the nodes and edges in 𝒟 independent of i
    for each connected component C of 𝒟_i do
        find the maximal number of loads, N, on any path through C
        for each load operation l in C do
            delay(l) ← delay(l) + delay(i) ÷ N
```

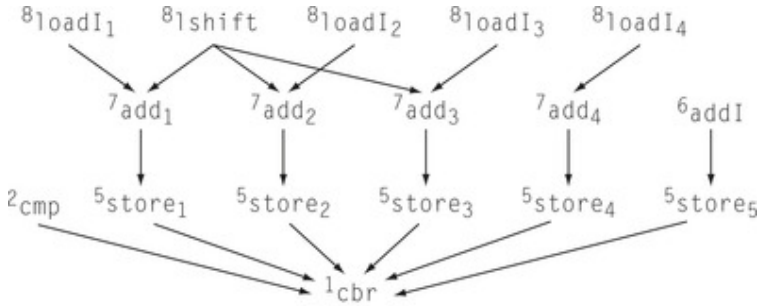
■ **FIGURE 12.7** Computing Delays for Load Operations.

Tie breaking

List scheduling is a greedy algorithm. To solve priority tie

- Number of immediate successors
- Total number of descendants
- The delay
- Number of operands

Forward vs backward list scheduling



Opcode	loadI	lshift	add	addI	cmp	store
Latency	1	1	2	1	1	4

	Integer	Integer	Memory
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	—	—

(a) Forward Schedule

	Integer	Integer	Memory
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	—	—

(b) Backward Schedule