CSc 553

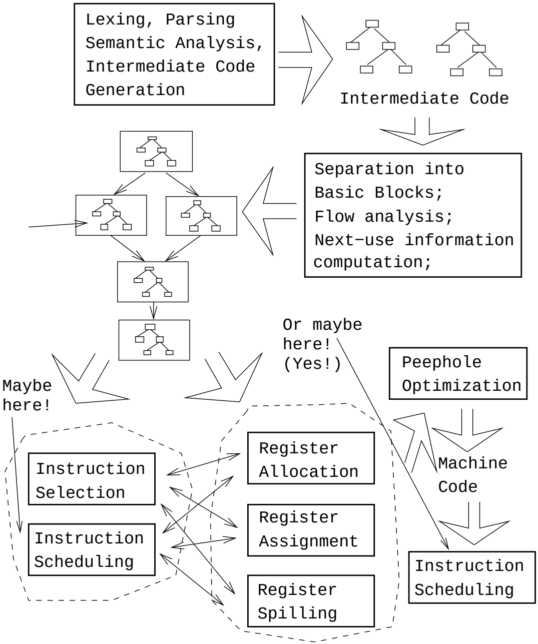
Principles of Compilation

24 : Instruction Scheduling I

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction



Lexing, Parsing Semantic Analysis, Intermediate Code Generation

Intermediate Code

Separation into Basic Blocks; Flow analysis; Next−use information computation;

Maybe here!

Or maybe here! (Yes!)

Peephole Optimization

Instruction Selection

Instruction Scheduling

Register Allocation

Register Assignment

Register Spilling

Machine Code

Instruction Scheduling

Instruction Scheduling

- Instruction scheduling is an inherently **machine dependent** task. Unlike other code improvements which work with high- or medium-level intermediate representations, instruction scheduling works explicitly with machine instructions.
- Instruction scheduling is often one of the last phases of code generation, performed after instruction selection and register allocation.
- However, scheduling affects register allocation, so, ideally, we'd like to do instruction scheduling and register allocation simultaneously.
- We'll look at a scheduling algorithm that works directly on the generated machine code.

Instruction Pipelines

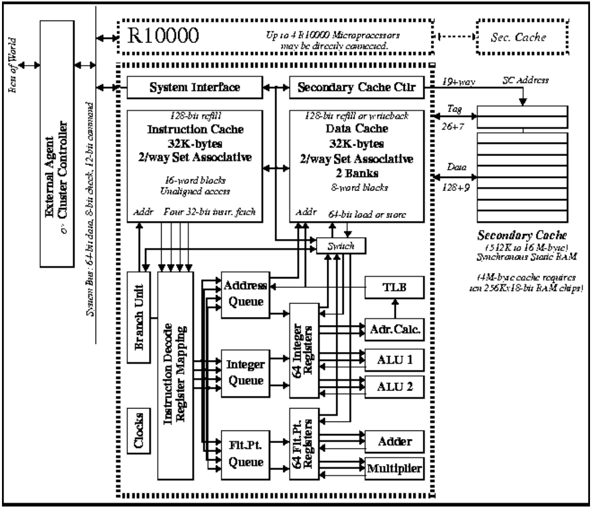
Instruction Pipelines

- Modern processors use instruction pipelines to speed up execution **throughput** (total amount of work done in a given time).
- The idea is that the execution of several instruction are overlapped in time, much the same as when a number of cars are assembled on an assembly line.
- The execution of instructions is broken down in several (4–7) individual pipeline stages.
- Pipelining doesn't speed up the execution of individual instructions; rather, it increases the number of simultaneously executing instruction and the rate at which they are started and completed.

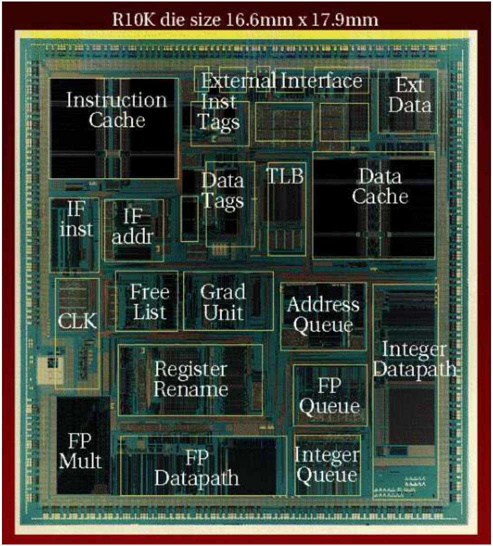
Instruction Pipelines II

- A typical 5-stage pipeline:
  (1) Fetch Fetch the instruction from memory (or instruction cache).
  (2) Decode Decode the instruction and fetch the values of registers that the instruction will need.
  (3) Execute Execute the instruction.
  (4) Memory Access memory (for load and store instructions).
  (5) Write Write the result of the instruction into the result register.
- Not all instructions make use of all the pipeline stages. There is often more than one execute stage.
- The MIPS R10000 has 7 stages: (1) Fetch, (2) Decode, (3) Issue Instruction and Read Registers, (4) Exec 1, (5) Exec 2, (6) Exec 3, (7) Register Write.

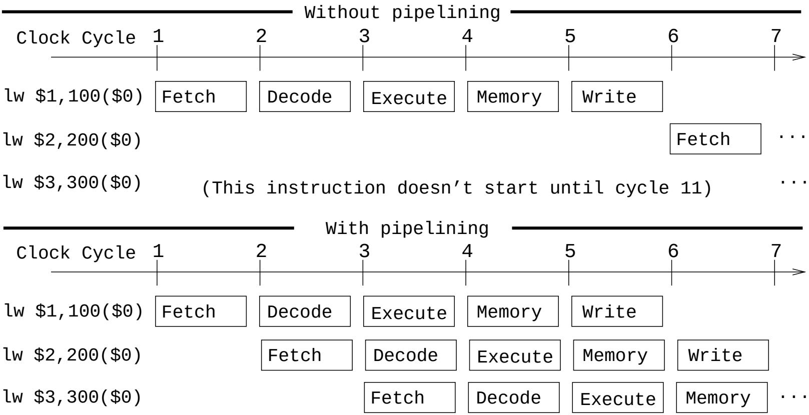
MIPS R10000 Block Diagram



MIPS R10000 Die Photo



Pipeline Example I



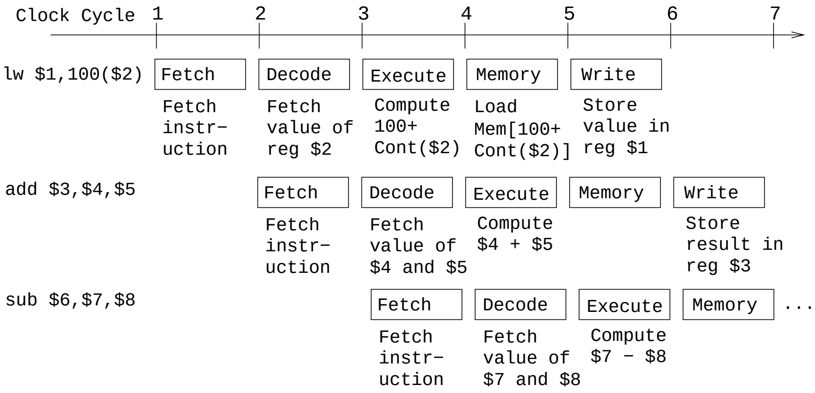
Without pipelining

Clock Cycle	1	2	3	4	5	6	7
lw $1,100($0)	Fetch	Decode	Execute	Memory	Write		
lw $2,200($0)						Fetch	···
lw $3,300($0)	(This instruction doesn't start until cycle 11)						···

With pipelining

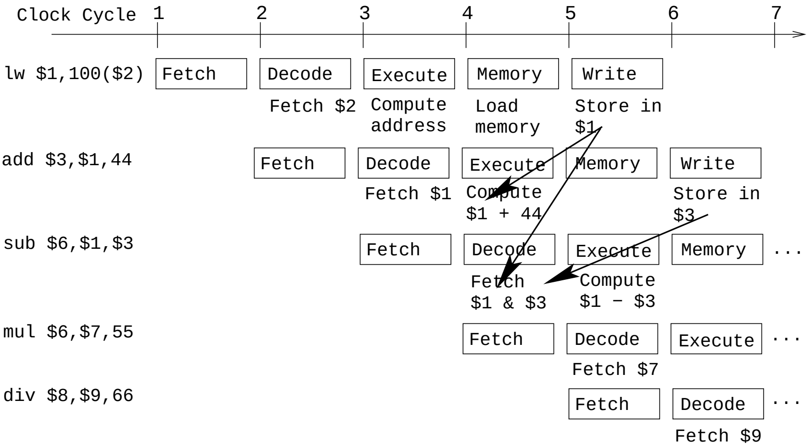
Clock Cycle	1	2	3	4	5	6	7
lw $1,100($0)	Fetch	Decode	Execute	Memory	Write		
lw $2,200($0)		Fetch	Decode	Execute	Memory	Write	
lw $3,300($0)			Fetch	Decode	Execute	Memory	···

Pipeline Example II



Clock Cycle	1	2	3	4	5	6	7
lw $1,100($2)	Fetch Fetch instruction	Decode Fetch value of reg $2	Execute Compute 100+ Cont($2)	Memory Load Mem[100+ Cont($2)]	Write Store value in reg $1		
add $3,$4,$5		Fetch Fetch instruction	Decode Fetch value of $4 and $5	Execute Compute $4 + $5	Memory	Write Store result in reg $3	
sub $6,$7,$8			Fetch Fetch instruction	Decode Fetch value of $7 and $8	Execute Compute $7 − $8	Memory	...

Pipeline Data Hazards I



Clock Cycle	1	2	3	4	5	6	7
lw $1,100($2)	Fetch	Decode Fetch $2	Execute Compute address	Memory Load memory	Write Store in $1		
add $3,$1,44		Fetch	Decode Fetch $1	Execute Compute $1 + 44	Memory	Write Store in $3	
sub $6,$1,$3			Fetch	Decode Fetch $1 & $3	Execute Compute $1 − $3	Memory	...
mul $6,$7,55				Fetch	Decode Fetch $7	Execute	...
div $8,$9,66					Fetch	Decode Fetch $9	...

Pipeline Data Hazards II

- The previous example shows one kind of problem we have with pipelines. Data hazards occur when the result of one instruction isn't ready in time for when that result is needed.
- Some processors will detect such situations and **stall** the pipeline until the value is ready. This is called a **harware/pipline interlock**.
- Interlocks are expensive (extra control logic on the chip which takes up valuable chip real estate) so some processors do away with them. Instead they rely on compilers and assemblers to insert `NOP`s when needed.

Pipeline Data Hazards III

- Example: On a MIPS 2000, the value loaded by a `ld` instruction isn't available to the immediately following instruction. The processor doesn't have hardware interlocks. If you give the following code the the assembler

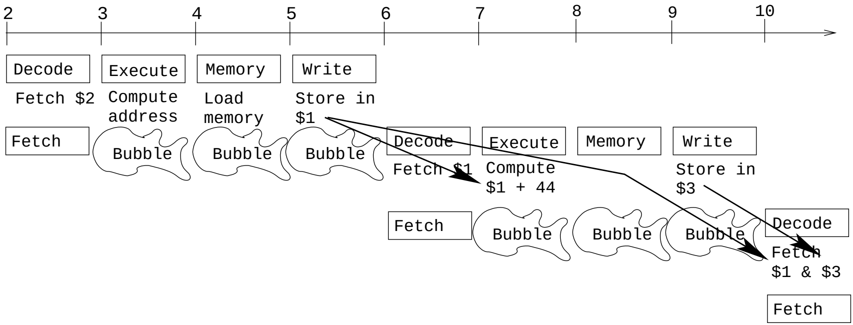
  ```
  ld        100($4), $5
  add       $5, $5, 56
  ```

  it will insert the necessary `NOP`s:

  ```
  ld        100($4), $5
  nop
  add       $5, $5, 56
  ```

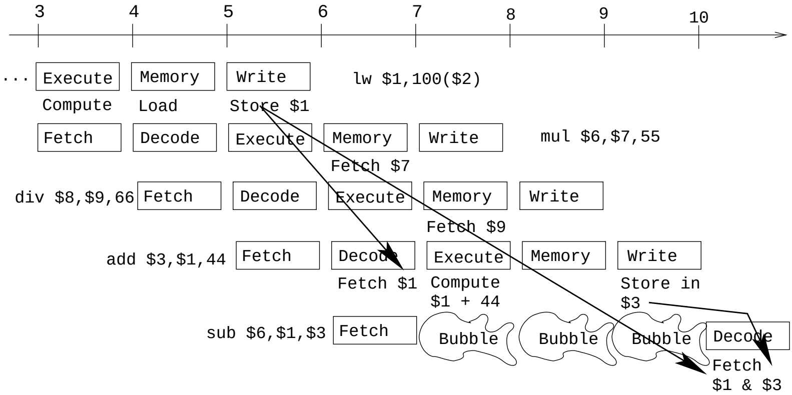
- Even if the hardware does have interlocks, the compiler (or assembler) should pay attention to the order in which instructions are scheduled. A well scheduled program may be upt o 50% faster than an naïvely scheduled one.

Pipeline Data Hazards – Stalls



- Some processors take care of data hazards themselves – the pipeline is **stalled** for a number of cycles until the hazard is resolved. This is like inserting one or more **bubbles** (NOPs) in the pipeline.

Pipeline Data Hazards – Reordering



3 4 5 6 7 8 9 10

... Execute Memory Write lw $1,100($2)

Compute Load Store $1

Fetch Decode Execute Memory Write mul $6,$7,55

Fetch $7

div $8,$9,66 Fetch Decode Execute Memory Write

Fetch $9

add $3,$1,44 Fetch Decode Execute Memory Write

Fetch $1 Compute $1 + 44 Store in $3

sub $6,$1,$3 Fetch Bubble Bubble Bubble Decode

Fetch $1 & $3

- The compiler can sometimes reorder instructions to avoid stalls.

Dependency Graphs

Dependency Graphs I

- An instruction scheduler reorders the instructions in an attempt at minimizing pipeline stalls.
- Obviously, we must not rearrange the code so that (when executed) it produces a different result from before!

```
add $3,$2,44
li  $2,66
```

$\xRightarrow{\text{Wrong!}}$

```
li  $2,66
add $3,$2,44
```

- We must therefore model all the dependencies between the instructions. We store this information in a directed **dependency graph**. The nodes of the graph are the instructions we're scheduling, and there's an edge $a \Rightarrow b$ if instruction a must come before instruction b.

Dependency Graphs II

- There can be three kinds of dependencies between instructions:

flow dependence

- Also, **true dependence** or **definition-use dependence**.

```
(i)        X   := ···
               .....
(j)        ··· := X
```

- Instruction `(i)` generates (**defines**) a value which is used by instruction `(j)`. We write (`i`) $\longrightarrow$ (`j`).

Dependency Graphs III

anti-dependence

- Also, **use-definition dependence**.

```
(i)        ··· := X
               .....
(j)        X   := ···
```

- Instruction `(i)` uses a value overwritten by instruction `(j)`. We write $(\mathtt{i}) \mathrel{+\!\!\!\!\longrightarrow} (\mathtt{j})$.

Dependency Graphs IV

Output-dependence

- Also, **definition-definition dependence**.

```
(i)        X   := ···            .....
(j)        X   := ···
```

- Instructions (i) and (j) both assign to (**define**) the same variable. We write (i)$-\!\!\ominus\!\!\rightarrow$(j).

- Regardless of the type of dependence, if instruction (j) depends on (i), then (i) has to be scheduled before (j).

Dependency Graphs V

Flow Dependence Example

```
(1)   ld    100($3), $2
            .....
(5)   add   $5, $2, 55
```

$\Rightarrow (1) \longrightarrow (5)$

- We can't move (1) after (5). If we did, $2 would not have the correct value at (5). Hence there is a flow dependence between (1) and (5).

Anti-Dependence Example

```
(2)   add   $5, $2, 55
            .....
(6)   li    $2, 33
```

$\Rightarrow (2) \longrightarrow\!\!\!\!\!\!\!\!\!+\;\;\;\; (6)$

- $2 gets a new value in (6). If we moved (2) after (6) $2 would have the wrong value in (6).

Dependency Graphs VI

Output-Dependence Example

```
(3)   li    $2, 33
(4)   add   $2, $5, 55   ⇒ (3)─⊖→(4)
(5)   add $6, $2, 44
```

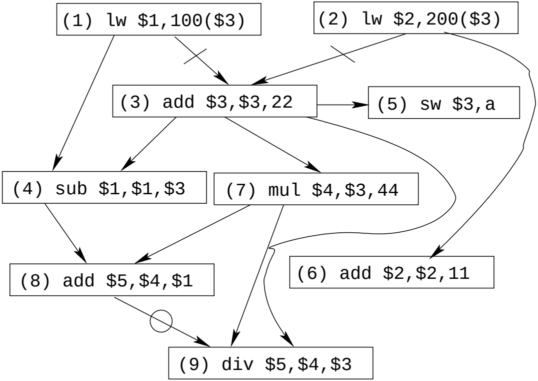
- If we switched (3) and (4), $2 would get the wrong value in (3).

Dependency Graphs VII

Complex Example

```
(1)   lw    $1, 100($3)
(2)   lw    $2, 200($3)
(3)   add   $3, $3, 22
(4)   sub   $1, $1, $3
(5)   sw    $3, 500($0)
(6)   add   $2, $2, 11
(7)   mul   $4, $3, 44
(8)   mul   $5, $4, $1
(9)   div   $5, $4, $3
```

Dependency Graphs VIII



- If instruction (b) depends on instruction (a) then (b) can be scheduled after a delay of one cycle.
- Real pipelines are more complex than this....

Topological Sorting

Topological Sorting I

- Problem: How to get dressed? I can't put my clothes on in an arbitrary order: socks have to be put on before shoes, shirt before tie, and so on.
- Each garment becomes a node in a DAG, and there's an edge from u to v if I have to put on u *before* v.

Topological Sorting:

"Order the nodes of the graph G in a sequence such that if $x \longrightarrow y$ is an edge is G then x comes before y in the ordering."

Simple Algorithm:

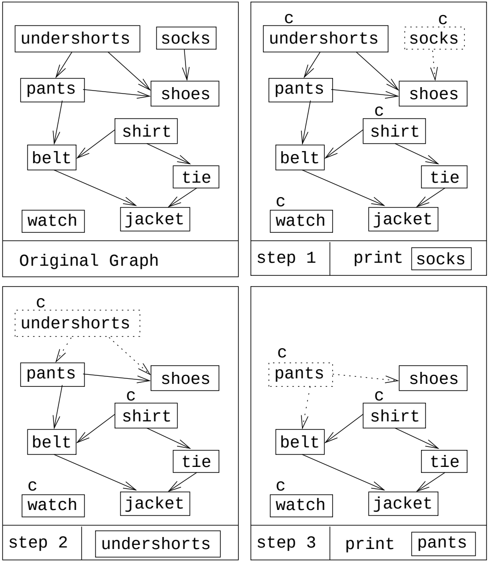
Repeat until no more nodes:

- Pick a node n without predecessors.
- Print n.
- Delete n and all its outgoing edges.

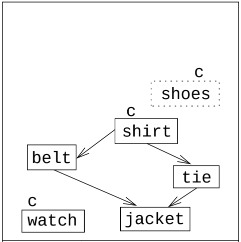
Topological Sorting II

- In the folloing example, candidates are marked with a c.
- When there is more than one candidate to chose from, we pick one at random.
- There are often many possible topological orders to chose from. For example, since the watch node has no dependencies at all, I can put it on at any time.

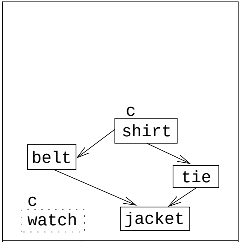
Topological Sorting III



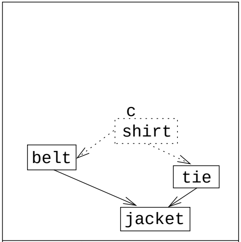
Topological Sorting IV



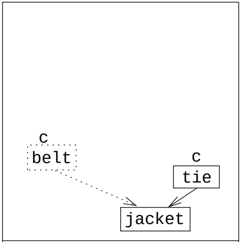
step 4 print shoes



step 5 print watch

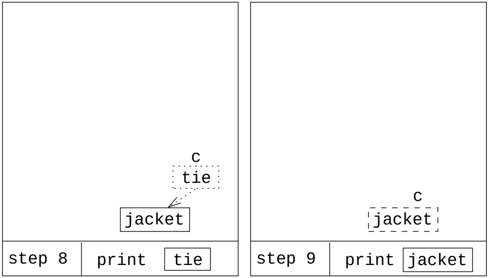


step 6 print shirt



step 7 print belt

Topological Sorting V



c

tie

jacket

step 8

print tie

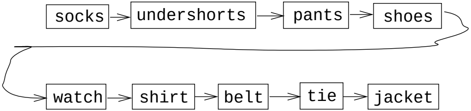
c

jacket

step 9

print jacket

Complete Topological Order



socks → undershorts → pants → shoes

watch → shirt → belt → tie → jacket

Algorithm

```
Last := {}; /* Last scheduled instr.  */
REPEAT
  Candidates := set of all nodes without
    predecessors (incoming edges);
  Colliding := set of all instructions that collide with
    the instructions in Last; i.e.  nodes $n$ for which
    there is an arch Last $\rightarrow$ $n$.
  realCandidates:= Candidates-Colliding;
  IF realCandidates $\neq$ {} THEN
    b := Use heuristic to pick the best
      realCandidate to schedule;
    Remove b and all its outgoing edges from the graph;
    print b; Last := {b};
  ELSE
    print NOP; Last := {};
  ENDIF
UNTIL Candidates = {};
```

Heuristic

- In the “standard” topological sorting algorithm we pick a node at random when there are several candidate nodes to chose from. In our scheduling algorithm we need a better heuristic.
- If there is more than one real candidate (a candidate that doesn’t collide with the last instruction scheduled) we choose one according to these criteria:
  1. Pick the candidate with the largest number of outgoing edges. Scheduling this instruction early will hopefully give us more freedom to schedule the remaining instructions.
  2. If all candidates have the same number of outgoing edges, pick the one that has the longest path to a leaf.

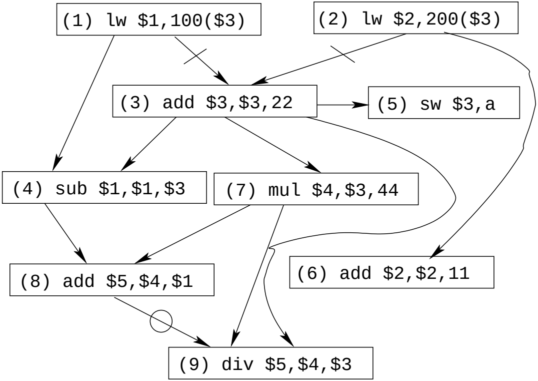
Example

Example I (a)

Complex Example

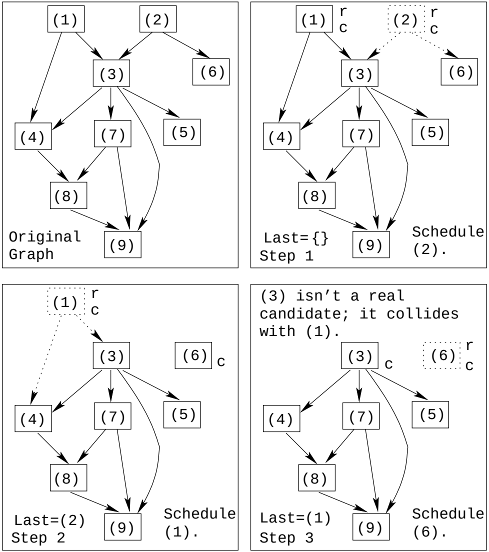
```
(1)    lw     $1, 100($3)
(2)    lw     $2, 200($3)
(3)    add    $3, $3, 22
(4)    sub    $1, $1, $3
(5)    sw     $3, 500($0)
(6)    add    $2, $2, 11
(7)    mul    $4, $3, 44
(8)    mul    $5, $4, $1
(9)    div    $5, $4, $3
```

Example I (b)



- If instruction `(b)` depends on instruction `(a)` then `(b)` can be scheduled after a delay of one cycle.
- We mark real candidates with an `r`.

Example I(c)



Original Graph

Last={}
Step 1

Schedule (2).

Last=(2)
Step 2

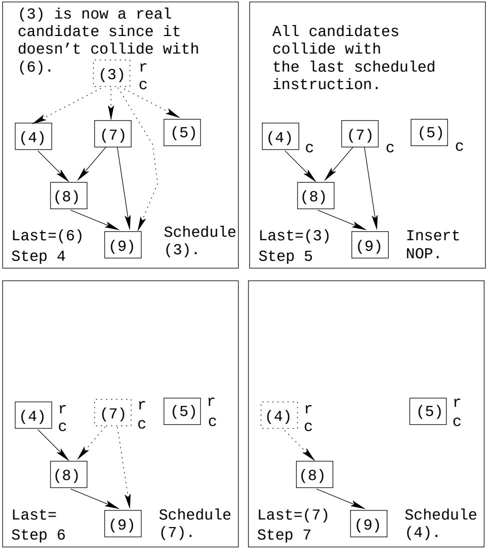
Schedule (1).

(3) isn't a real candidate; it collides with (1).

Last=(1)
Step 3

Schedule (6).

Example I(d)



(3) is now a real candidate since it doesn't collide with (6).

Last=(6)
Step 4

Schedule (3).

All candidates collide with the last scheduled instruction.

Last=(3)
Step 5

Insert NOP.

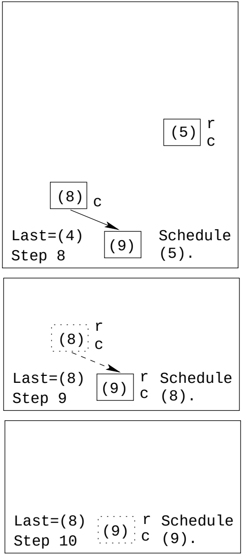
Last=
Step 6

Schedule (7).

Last=(7)
Step 7

Schedule (4).

Example I(e)



Last=(4)
Step 8

Schedule (5).

Last=(8)
Step 9

Schedule (8).

Last=(8)
Step 10

Schedule (9).

Generated Code

```
(2) lw $2,200($3)

(1) lw $1,100($3)
(6) add $2,$2,11
(3) add $3,$3,22
    NOP
 (7) mul $4,$3,44
 (4) sub $1,$1,$3
(5) sw $3,a
 (8) add $5,$4,$1
 (9) div $5,$4,$3
```

NOTE:
(8) ⟶ (9)
is an output dependence. Hence there's no need for a NOP between them.

Summary

Readings and References

- Read the Tiger book, Chapter 20, pp. 474–497.
- For a background on pipelining and hazards, see Patterson & Hennessy, “Computer Organization and Design – The Hardware/Software Interface”, pp. 364–367,
- Wilhelm & Maurer, “Compiler Design”, pp. 557–558, 570–580.
- Steven Muchnick, *Advanced Compiler Design and Implementation*, Section 9.2, pp. 269–274 and Chapter 17, pp. 531–547.

Summary

- Some processors have instructions with pipeline **hot-spots**: A value that is computed in one cycle must be consumed in the next cycle (before it is overwritten).
- We have assumed that the delay between two instructions is always exactly 1 (one) cycle. Real processors require different delays depending on the instruction.

Homework

Homework I

- Consider the following basic block:

```
(1)   lw    $1, 100($2)
(2)   li    $3, 100
(3)   add   $2, $3, 55
(4)   sub   $1, $3, 66
(5)   li    $5, 99
(6)   add   $7, $3, $5
(7)   li    $3, 93
(8)   add   $9, $2, $7
(9)   sub   $1, $9, $3
```

1. Construct the dependency graph.
2. Apply the scheduling algorithm to the graph. Assume that there is a one-cycle delay between an instruction and when it's computed value can be used.
3. Is the generated schedule an optimal one?

Exam Problems

Exam Problem I (415.730 '96)

1. What is a **pipeline data hazard**?
2. Why is it important for the compiler to perform aggressive instruction scheduling, even when the architecture has **hardware interlocks**?
3. Describe the algorithms and data structures needed to perform instruction scheduling in the presence of pipeline data hazards.