Compiler Construction

Dependencies and RISC pipelines

Why and When reordering?

Goal

Reorder the instructions within each basic block

- preserves the dependencies between those instructions (and hence the correctness of the program)
- obtains the best performances

A word on Dependencies 1/4

Two instructions are independent if they can be permuted without altering the consistency

A word on Dependencies 2/4

- The 3 following instructions are independent

$$\begin{aligned} \text{inst}_1 &: \quad a \leftarrow 42 \\ \text{inst}_2 &: \quad b \leftarrow 51 \\ \text{inst}_3 &: \quad c \leftarrow 0 \end{aligned}$$

- inst_1, inst_2 and $\textit{inst}_3$ can then be reordered

$$\begin{aligned} \text{inst}_1 &: \quad a \leftarrow 42 \\ \text{inst}_2 &: \quad b \leftarrow 51 \\ \text{inst}_3 &: \quad c \leftarrow 0 \end{aligned} \quad \left\| \quad \begin{aligned} \text{inst}_1 &: \quad a \leftarrow 42 \\ \text{inst}_3 &: \quad c \leftarrow 0 \\ \text{inst}_2 &: \quad b \leftarrow 51 \end{aligned} \right\| \quad \begin{aligned} \text{inst}_3 &: \quad c \leftarrow 0 \\ \text{inst}_1 &: \quad a \leftarrow 42 \\ \text{inst}_2 &: \quad b \leftarrow 51 \end{aligned}$$

$$\begin{aligned} \text{inst}_1 &: \quad c \leftarrow 0 \\ \text{inst}_2 &: \quad b \leftarrow 51 \\ \text{inst}_3 &: \quad a \leftarrow 42 \end{aligned} \quad \left\| \quad \begin{aligned} \text{inst}_1 &: \quad b \leftarrow 51 \\ \text{inst}_3 &: \quad c \leftarrow 0 \\ \text{inst}_2 &: \quad a \leftarrow 42 \end{aligned} \right\| \quad \begin{aligned} \text{inst}_3 &: \quad b \leftarrow 51 \\ \text{inst}_1 &: \quad a \leftarrow 42 \\ \text{inst}_2 &: \quad c \leftarrow 0 \end{aligned}$$

A word on Dependencies 3/4

Two instructions are dependant if the first one needs to be executed before the second one.

A word on Dependencies 4/4

- The 3 following instructions are dependent, i.e. no reordering is possible!

$$\begin{aligned} \text{inst}_1 : &\quad a \leftarrow 42 \\ \text{inst}_2 : &\quad b \leftarrow a + 51 \\ \text{inst}_3 : &\quad c \leftarrow b \times 12 \end{aligned}$$

- Two kind of dependencies:
  - **Data dependencies**: the instruction manipulates a ”variable” computed by another instruction.
  - **Instruction dependencies**: the instruction is a ”cjump”, the next instruction depends of the ”cjump”.

Read after Write (RAW)

An instruction reads from a location after an earlier instruction has written to it.

```
inst1 : lw   $2, 0($4)
inst2 : addi $6, $2, 42
```

$inst_1$ and $inst_2$ cannot be permuted, otherwise $inst_2$ would read an old value from $2.

Write after Read (WAR)

An instruction writes to a location after an earlier instruction has read from it.

```
inst1 : lw   $2, 0($4)
inst2 : addi $4, $12, 42
```

$inst_1$ and $inst_2$ cannot be permuted, otherwise $inst_1$ would read a new value for $4

Write after Write (WAW)

An instruction writes to a location after an earlier instruction has written to it.

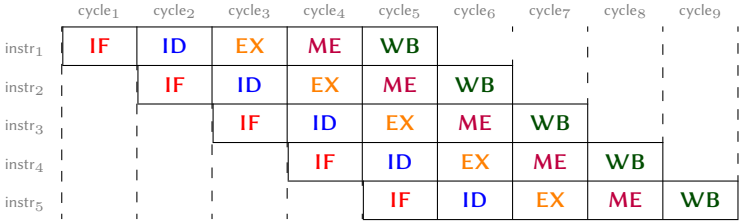
inst_1 : `add $1, $2, $3`
inst_2 : `add $1, $5, $6`

inst_1 and inst_2 cannot be permuted, otherwise inst_1 would write an old value in $1

Instructions Pipeline

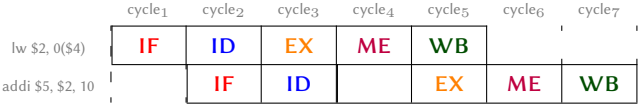
The microprocessor (MIPS) contains 5 stages:

- IF: Instruction Fetch
- ID: Instruction Decode. Read operands from registers, compute the address of the next instruction
- EX Execute instructions requiring the ALU
- ME Read/write into Memory
- WB Write Back. Results are written into registers.



Hazard: RAW dependencies 1/2

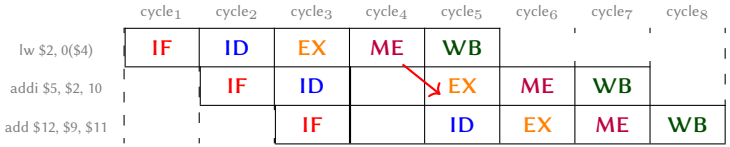
Some instruction requires a result computed by a previous one!



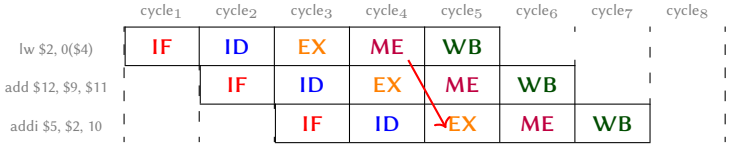
- lw produces its result into $2 during the ME stage
- ADDI requires $2 for the EX stage
- In this example, 1 stall (cycle 4)

Hazard: RAW dependencies 2/2

Consider now the following example:



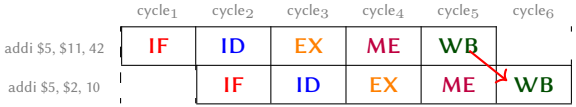
Instruction 3 is independent from the others so we can change the order!



Hazard: WAW dependencies

Two instructions write in the same register!

Consider the following example:

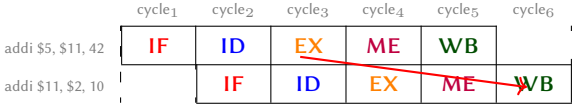


WAW do not produce stalls !
(even when writing in the same memory address)

Hazard: WAR dependencies

One instruction writes where a previous one reads!

Consider the following example:



WAR do not produce stalls !

Summary