UT 2. Pipelined Computers

Lecture 2.2 Static instruction scheduling

J. Duato, J. Flich, P. López, V. Lorente,
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Department of Computer Engineering
Universitat Politècnica de València

Contents

Bibliography


John L. Hennessy and David A. Patterson.
Computer Architecture, Fifth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012.

Contents

Problem

- Some integer instructions perform complex computations (`mult, div`).
- Floating point instructions (`add.s, add.d, mult.s, div.s,...`)

$\rightarrow$ they need more time at the EX stage $\Rightarrow$ How to pipeline the instruction unit in the presence of such operations?

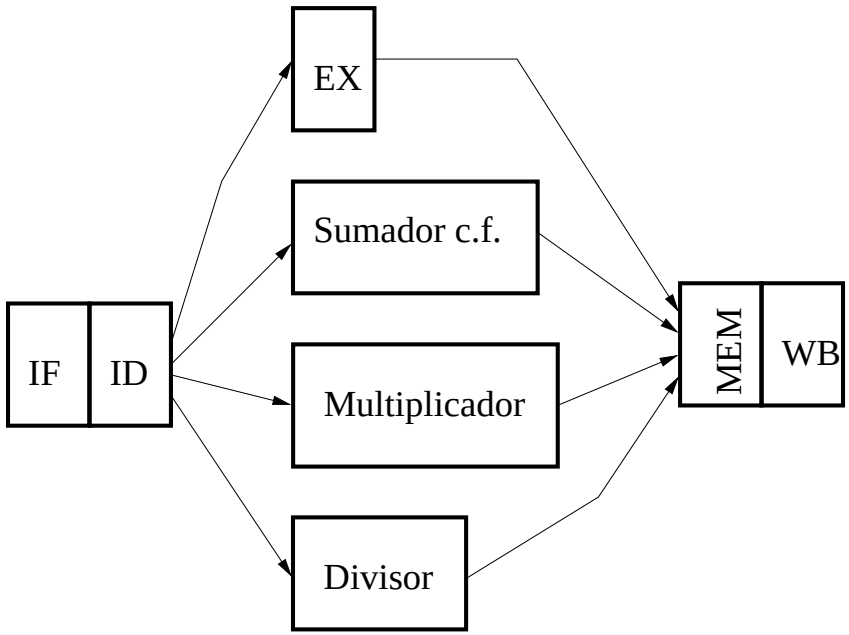
Solutions

- Increase the clock period of the instruction unit $\rightarrow$ the machine slows down !
- Add more *hardware* to keep EX taking the *same time* required by simple instructions $\rightarrow$ cost $\uparrow$ and it is not always possible.
- Enable EX stage of complex operations to take several clock cycles (**multicycle** operations) $\rightarrow$ the clock period does not change.
  Example: `mult` with a 40 ns multiplier

  Stages: IF ID EX EX EX EX MEM WB

Multiple operators

- As new instructions require specific *hardware*, new *specialized* operators are added instead of a single multifunction operator.
- During ID stage, the instruction is issued to the appropriate operator
- Once the operation ends, it is sent to the MEM stage.



Types of multicycle operators

New multicycle operators can be either conventional or pipelined.
$\rightarrow$ Typical parameters:

- latency or evaluation time (time required to obtain the first result), and
- initiation rate *IR* (reciprocal of the time between results).
  if it is pipelined, $IR = 1$

Example of operators added to the MIPS

- Add./sub., floating point, pipelined. T_{ev}: 4. IR: 1 every cycle.
- Mult., integer/floating point, pipelined. T_{ev}: 7. IR: 1 every cycle.
- Div., integer/floating point, non-pipelined. T_{ev}: 24. IR: 1 every 24 cycles.

Example of execution with multicycle operations

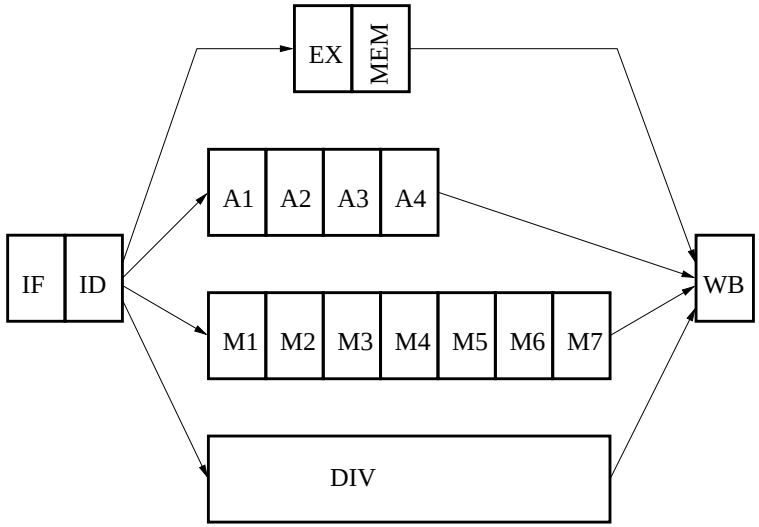
```
DADD R1,R2,R3   IF ID EX ME WB
ADD.D F0,F2,F4     IF ID A1 A2 A3 A4 ME  WB
MULT.D F6,F8,F10      IF ID M1 M2 M3  M4  M5  M6  M7  ME  WB
MULT.D F12,F14,F16       IF ID M1 M2  M3  M4  M5  M6  M7  ME  WB
DIV.D F18,F20,F22           IF ID DIV DIV DIV DIV DIV DIV DIV DIV ...
```

Review of the pipelined instruction unit with multicycle operators:

- Not all instructions take the same number of clock cycles
- MEM stage is empty in multicycle operations

$\Rightarrow$ Deletion of MEM stage from the datapath of multicycle operations

Pipelined MIPS including new operators



- New inter-stage registers are required:
  1. ID/EX $\rightarrow$ ID/EX, ID/A1, ID/M1, ID/DIV
  2. A1/A2, A2/A3, . . . , A3/A4
  3. M1/M2, M2/M3, . . . , M6/M7
- A multiplexor is required to handle WB stage inputs.
- Unnecessary structural hazards are avoided by ending the execution of stores in MEM.

Structural hazards for using units with IR $<$ 1 op/cycle (1/2)

Example:

i1 DIV.D ... (Tev = 6 cycles, IR = 1 op/6 cycles)
i2 DIV.D ...
i3 Integer instruction
i4 Integer instruction

Problem: Two instructions simultaneously in the same operator!

i1	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	WB	
i2		IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	WB
i3			IF	ID	EX	ME	WB			
i4				IF	ID	EX	ME	WB		

Structural hazards for using units with IR $<$ 1 op/cycle (2/2)

Solution:
The second instruction must wait at ID, by inserting stalls, until the operator becomes free.

i1	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	WB				
i2		IF	ID	*ID*	*ID*	*ID*	*ID*	*ID*	DIV	DIV	DIV	DIV	...
i3			IF	*IF*	*IF*	*IF*	*IF*	*IF*	ID	EX	M	WB	
i4									IF	ID	EX	M	

$\Rightarrow$ ID concentrates all the detection logic
$\Rightarrow$ As soon as one instruction is stalled at ID, no further instructions are fetched (IF)

Struct. hazards derived from simultaneous register file writing

Example:

i1 MUL.D ... (Tev = 4 cycles, IR = 1 op/cycle)
i2 ADD.D ... (Tev = 3 cycles, IR = 1 op/cycle)
i3 ADD.D ...
i4 Integer instructions

Problem: Several instructions in WB!

i1	IF	ID	M1	M2	M3	M4	**WB**	
i2		IF	ID	A1	A2	A3	**WB**	
i3			IF	ID	A1	A2	A3	**WB**
i4				IF	ID	EX	ME	**WB**

$\rightarrow$ reduce the number of simultaneous accesses to the register file during WB: separate integer from floating point register files.

Independent register files for floats and integers

Goal: integer and floating point instructions use *different* reg. files.

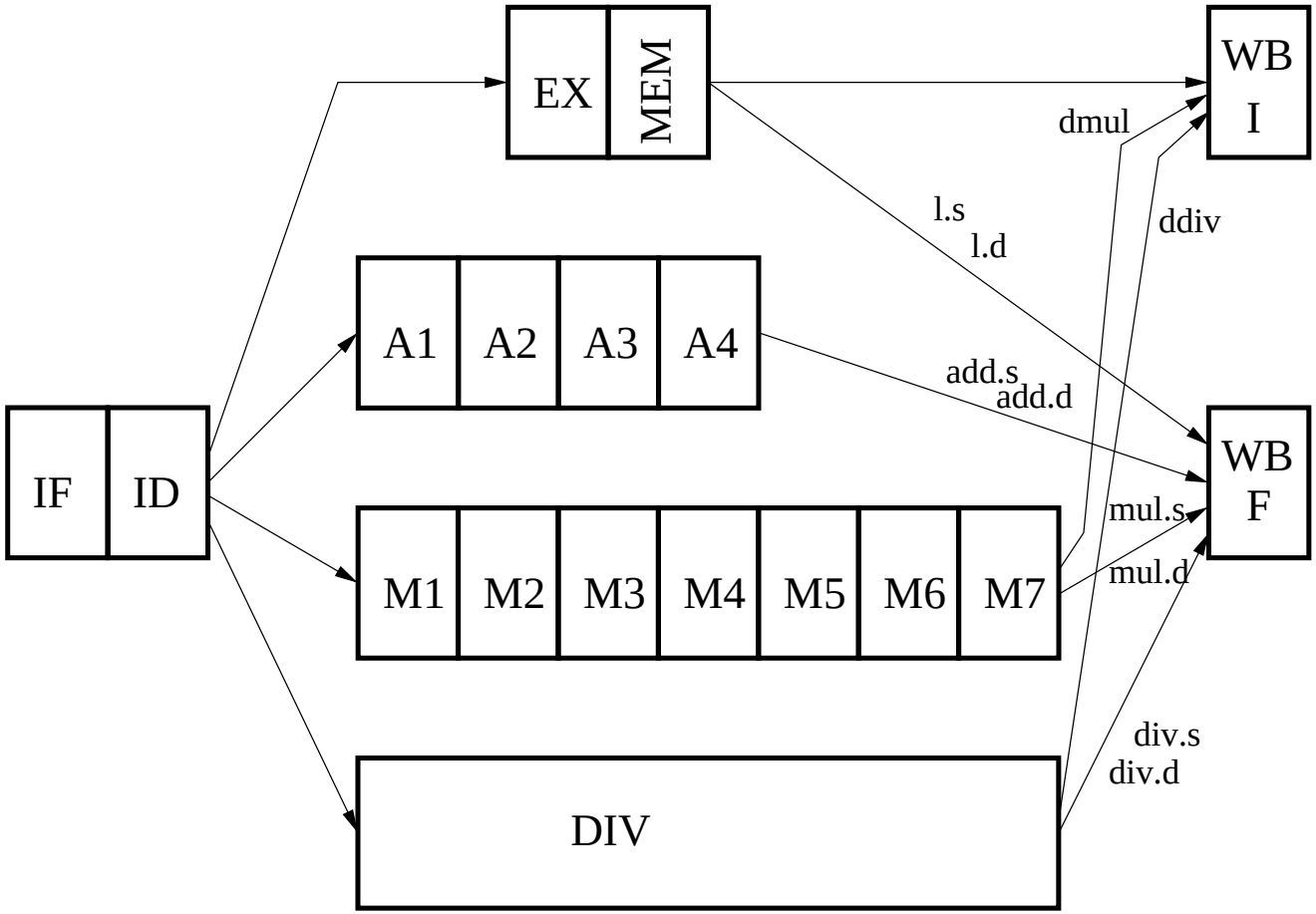
Advantages

- Structural hazards are reduced.
- The total number of registers is doubled, without making decoding logic more complex, without increasing access time and without adding more bits to the format.
- Register file bandwidth is doubled, without adding more ports or increasing the access time.

Drawbacks

- Sometimes, it is necessary to exchange information between both register files (`MFC0`, `MTC0` and `MFC1`, `MTC1` instructions).
- The number of registers of each type is limited a priori.

Pipelined MIPS with two independent register files



Struct. hazards from a simultaneous register file writing (1/2)

Problem:

i1 MUL.D ... (Tev = 4 cycles, IR = 1 op/cycle)
i2 ADD.D ... (Tev = 3 cycles, IR = 1 op/cycle)
i3 ADD.D ...
i4 Integer instruction

i1	IF	ID	M1	M2	M3	M4	**WB**	
i2		IF	ID	A1	A2	A3	**WB**	
i3			IF	ID	A1	A2	A3	WB
i4				IF	ID	EX	ME	WB

Struct. hazards from a simultaneous register file writing (2/2)

Solutions:

- Increase the number of ports $\rightarrow$ *On average* only one writing operation is carried out per cycle (since only one instruction is issued per cycle and there is only one writing action per instruction) $\rightarrow$ this is not efficient.
- Stall insertion $\rightarrow$ ID stage checks whether an instruction writes to the register file at the same time that another one does.

i1	IF	ID	M1	M2	M3	M4	WB		
i2		IF	ID	*ID*	A1	A2	A3	WB	
i3			IF	*IF*	ID	A1	A2	A3	WB
i4					IF	ID	EX	ME	WB

$\Rightarrow$ Detection logic concentrated at ID.

$\Rightarrow$ Stalls at ID prevent fetching new instructions (IF).

Structural hazards (1/2)

RAW (Read After Write) hazards happen when an instruction produces a result required by a following one.

Example:

i1	ADD.D **F0**, F2, F4
i2	MUL.D F6, **F0**, F8
i3	Integer instruction
i4	Integer instruction
i5	Integer instruction

Structural hazards (2/2)

Solution: stall insertion until a short-circuit can be applied.

ADD.D **F0**,...	IF	ID	A1	A2	A3	A4	WB			
MUL.D ...,**F0**		IF	ID	*ID*	*ID*	*ID*	M1	M2	M3	...
i3			IF	*IF*	*IF*	*IF*	ID	EX	ME	WB
i4							IF	ID	EX	ME
i5								IF	ID	EX

$\Rightarrow$ Detection logic concentrated at ID
$\Rightarrow$ Stalls at ID prevent instruction fetching (IF)

With multicycle operations, the execution stage may take several clock cycles $\rightarrow$ penalty (number of *stalls*) can be very important.

WAW data hazards (1/3)

WAW (Write After Write) hazards happen when two close instructions write to the same register.

1. In addition to the WAW hazard, there is a RAW hazard.

   i1 MUL.D **F0**,F2,F4
   i2 DIV.D F6,**F0**,F8
   i3 ADD.D **F0**,F10,F12

   RAW hazard solving, by inserting stalls at the ID stage, also solves the WAW hazard:

i1	MUL.D **F0**,F2,F4	IF	ID	M1	M2	M3	M4	M5	M6	M7	**WB**	
i2	DIV.D F6,F0,F8		IF	ID	*ID*	*ID*	*ID*	*ID*	*ID*	*ID*	DIV	...
i3	ADD.D **F0**,F10,F12			IF	*IF*	*IF*	*IF*	*IF*	*IF*	*IF*	ID	A1

WAW data hazards (2/3)

2. Example 2: There is only a WAW hazard

i1 MUL.D **F0**,F2,F4
i2 ADD.D **F0**,F10,F12

Problem: the order of write operations is not correct. F0 register keeps the intermediate value, but not the final one.

i1	MUL.D **F0**,F2,F4	IF	ID	M1	M2	M3	M4	M5	M6	M7	**WB**
i3	ADD.D **F0**,F10,F12		IF	ID	A1	A2	A3	A4	**WB**		

Solution: Detection at ID and stall insertion.

i1	MUL.D **F0**,F2,F4	IF	ID	M1	M2	M3	M4	M5	M6	M7	**WB**	
i3	ADD.D **F0**,F10,F12		IF	ID	*ID*	*ID*	*ID*	A1	A2	A3	A4	**WB**

Although a good compiler should not generate code with two write operations to the same register without any intermediate read.

WAW data hazards (3/3)

... there are *unexpected* situations where the following may occur:

- A hazard between an instruction of the delay-slot and a following one. Example:

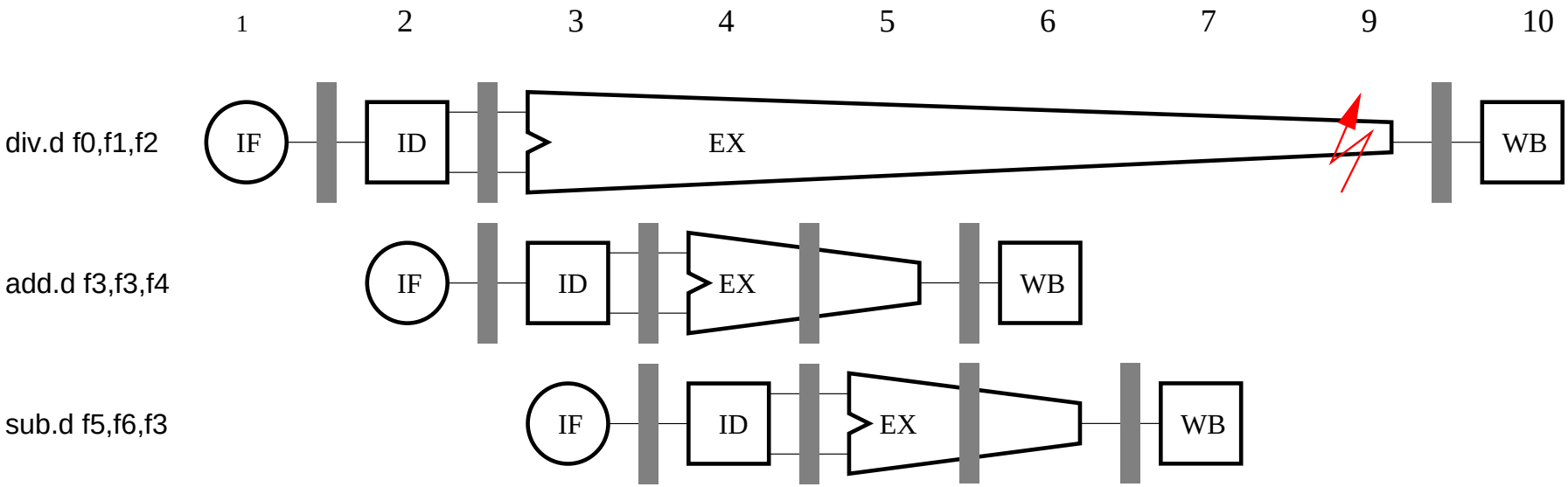
```
        BNEZ R1,label
        DIV.D F0,F2,F4 ; Delay-slot filled with a
                       ; following instruction
                       ; since it is normally not taken %
        ....
label:L.D F0,mem(R1)
```

  If the branch is taken, there is a WAW hazard between `DIV.D` and `L.D`

Exception management

Problem:

The presence of multicycle operations may alter the instruction execution order → when an exception occurs, some of the instructions *following* the one triggering the exception may have already finished:



Contents

Instruction level parallelism

The problem:

Pipelined unit without multicycle operations $\longrightarrow$ few stalls $\longrightarrow$ CPI ≈ 1

Pipelined unit with multicycle operations $\longrightarrow$ many stalls $\longrightarrow$ CPI $>> 1$

Instruction Level Parallelism or **ILP**:

- Potential overlapping in the execution of instruction sequences.
- Relies on the independence among the considered instructions.

$$\textbf{ILP} \uparrow \longrightarrow \text{few conflicts} \longrightarrow \text{few stalls} \longrightarrow \textbf{CPI} \downarrow$$

Dependencies between program instructions

Two instructions are independent if they can be simultaneously executed without any problem $\Leftrightarrow$ They can be reordered.

Types of dependencies:
- Data dependency
- Name dependency
- Control dependency

Data dependency

Given two instructions *i* and *j*, *j* logically after *i*: $\left\{ \begin{array}{l} \text{instr } i \\ \ldots \\ \text{instr } j \end{array} \right.$

a data dependency exists if:

- *i* produces a result used by *j*
- There exist a data dependency between *i* and *k*, and *k* produces a result used by *j*. This chain can be as long as the program is.

Example:

```
loop:  L.D F0,0(R1)
       ADD.D F4,F0,F2
       S.D F4,0(R1)
       ...
```

Shared data causing the dependency can be stored in either registers or memory.

Name dependency (1/2)

They occur when two instructions use the same register or memory location, but there is no data flow between them.

Given two instructions *i* and *j*, *j* logically after *i*: $\begin{cases} \text{instr } i \\ \ldots \\ \text{instr } j \end{cases}$

the following name dependencies may occur:

1. Anti-dependency. An anti-dependency occurs when an instruction requires a value that is later updated. Example: instruction *j* writes on a register or memory location being read by instruction *i*.
2. Output dependency. An output dependency occurs when the ordering of instructions will affect the final output value of a variable. Example: instructions *i* and *j* write on the same register or memory location.

Name dependency (2/2)

Example:

```
loop:  L.D  F0,0(R1)
       ADD.D F4,F0,F2
       S.D  F4,0(R1)
       L.D  F0,-8(R1)
       ADD.D F4,F0,F2
       S.D  F4,-8(R1)
       ...
```

Control dependencies (1/2)

An instruction is control dependent on a preceding instruction if the outcome of the latter determines whether the former should be executed or not.

Every instruction in a program (except those at the beginning of the program) has a control dependency with some branch.

Example:

```
        ...
        BEQZ R1, exit
        L.D F10,0(R1)
        ADD.D F14,F10,F2
        S.D F14,0(R1)
        DSUB R1,R1,#8
        ...
exit:
```

Control dependencies (2/2)

The problem

ILP $\downarrow$ $\longrightarrow$ Simultaneous presence of dependent instructions in the pipelined unit $\longrightarrow$ (potential) hazards $\longrightarrow$ (potential) stalls $\longrightarrow$ CPI $\uparrow$

$\Downarrow$

Type of Dependency	*Hazard:*
Data dependency	RAW
Anti-dependency	WAR
Output dependency	WAW
Control dependency	Control hazard

Contents

Basic block concept

The goal is to increase the ILP of instructions currently under execution in the pipelined unit.

Sequences of instructions between branch instructions are named **basic blocks**. We need to determine the amount of parallelism that can be extracted from the execution of the instructions in each basic block.

```
        beqz r1, L
        instr
        instr
Basic   instr
block   ...
        beqz r1, L
```

Is there enough ILP in a single basic block?

1. Statistics: 15% of instructions are branches $\rightarrow$ 6 or 7 instructions per basic block.
2. Instructions in a basic block usually exhibit dependencies among them.

Solution $\rightarrow$ Exploit the existing ILP among multiple basic blocks: instructions from different basic blocks are executed in parallel.

A particular case is the one concerning *loop-level parallelism*, which exploits the ILP existing among loop iterations by overlapping them:

```
for i := 1 to 1000 do
    x[i] := x[i] + s;
```

How do we increase the ILP by overlapping basic blocks?

Static instruction scheduling

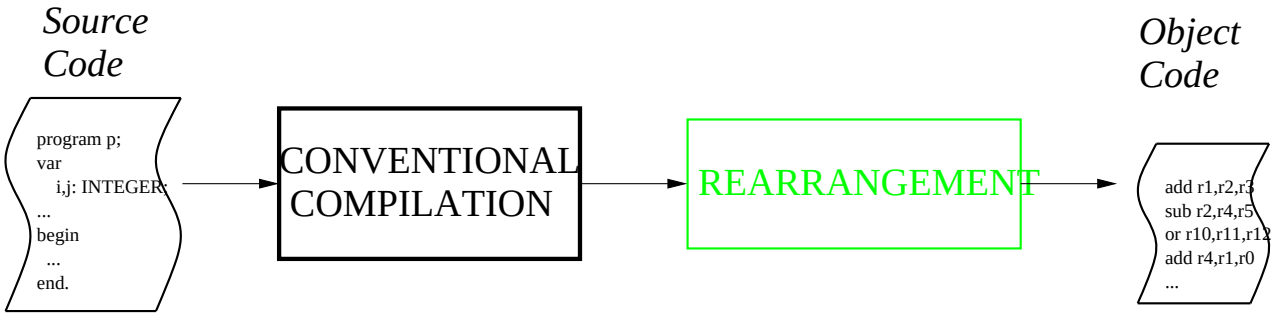
The compiler reorders/modifies the code.

Dynamic instruction scheduling

Hardware reorders instructions at runtime.

Static instruction scheduling

The compiler reorders/modifies the code to increase ILP by reducing/deleting existing dependencies and their effects (hazards and stalls):



Some static instruction scheduling techniques:

Loop unrolling Moves dependent instructions away from each other and exploits ILP across several basic blocks.

Software pipelining Reorders the code in order to move dependent instructions away from each other.

Contents

Example: $\vec{Z} = a + \vec{Y}$ (1/2)

```
        i = 0;
        while (i<n) {
          z[i] = a + y[i];
          i = i+1;
        }
```

↓

```
start:
        daddi r1, r0, y      ; r1 = y address
        daddi r2, r0, z      ; r2 = z address
        l.d f0, a(r0)        ; f0 = a
        daddi r3, r1, #512   ; 64 elements are 512 bytes
loop:
        l.d f2, 0(r1)        ; L
        add.d f4, f0, f2     ; A
        s.d f4,0(r2)         ; S
        daddi r1, r1, #8
        daddi r2, r2, #8
        dsub r4, r3, r1
        bnez r4, loop
```

Example: $\vec{Z} = a + \vec{Y}$ (2/2)

Data:

1 Floating point adder (Tev = 4 cycles, IR = 1 op/cycle)

$\rightarrow$ 4 stalls are inserted in each loop iteration:

`l.d f2,0(r1)`	IF	ID	EX	M	WB						
`add.d f4,f0,f2`		IF	ID	*ID*	A1	A2	A3	A4	WB		
`s.d f4,0(r2)`			IF	IF	ID	*ID*	*ID*	*ID*	EX	M	
`dadd r1,r1,#8`					IF	IF	IF	IF	ID	EX	M
`dadd r2,r2,#8`									IF	ID	EX
`dsub r4,r3,r1`										IF	ID
`bnez r4,loop`											IF

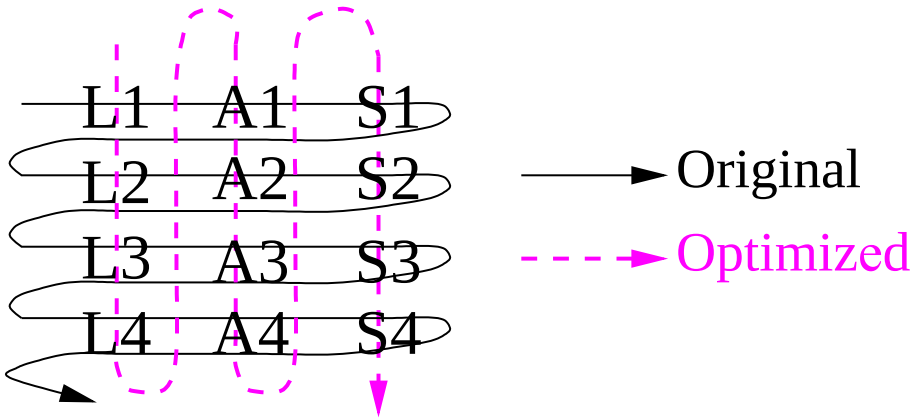
Basic idea (1/2)

The loop code is replicated several times, thus reducing the number of iterations to be executed.

```
i = 0;
while (i<n) {
   z[i] = a + y[i];
   z[i+1] = a + y[i+1];
   z[i+2] = a + y[i+2];
   z[i+3] = a + y[i+3];
   i = i+4;
} /* endwhile */
```

Basic idea (2/2)

- It reduces the overhead produced by loop control instructions.
- Increasing the size of the basic block increases the freedom of the compiler to separate instructions exhibiting data dependencies:



Loop code replicated 4 times:

```
start:
        daddi r1, r0, y       ; r1 = y address
        daddi r2, r0, z       ; r2 = z address
        l.d f0, a(r0)         ; f0 = a
        daddi r3, r1, #512    ; 64 elem.  are 512 bytes
loop:
        l.d f2, 0(r1)         ; (1.1)
        add.d f4, f0, f2      ; (1.2)
        s.d f4,0(r2)          ; (1.3)
        l.d f2, 8(r1)         ; (2.1)
        add.d f4, f0, f2      ; (2.2)
        s.d f4,8(r2)          ; (2.3)
        l.d f2, 16(r1)        ; (3.1)
        add.d f4, f0, f2      ; (3.2)
        s.d f4,16(r2)         ; (3.3)
        l.d f2, 24(r1)        ; (4.1)
        add.d f4, f0, f2      ; (4.2)
        s.d f4,24(r2)         ; (4.3)
        daddi r1, r1, #32     ; 4 times 8 = 32
        daddi r2, r2, #32
        dsub r4, r3, r1
        bnez r4, loop
```

Optimization:

It is necessary to perform *register renaming* to eliminate name dependencies:

```
loop:   l.d f2, 0(r1)
        add.d f4, f0, f2
        s.d f4,0(r2)
        l.d f2, 8(r1)
        add.d f4, f0, f2
        s.d f4,8(r2)
        ...
```

→

```
loop:   l.d f2, 0(r1)
        add.d f4, f0, f2
        s.d f4,0(r2)
        l.d f12, 8(r1)
        add.d f14, f0, f12
        s.d f14,8(r2)
        ...
```

Loop code replicated 4 times and *optimized:* (1/2)

```
start:
        dadd r1, r0, y        ; r1 = y address
        dadd r2, r0, z        ; r2 = z address
        l.d f0, a(r0)         ; f0 = a
        dadd r3, r1, #512     ; 64 elem.  are 512 bytes
loop:
        l.d f2, 0(r1)         ; (1.1)
        l.d f6, 8(r1)         ; (2.1)
        l.d f10, 16(r1)       ; (3.1)
        l.d f14, 24(r1)       ; (4.1)
        add.d f4, f0, f2      ; (1.2)
        add.d f8, f0, f6      ; (2.2)
        add.d f12, f0, f10    ; (3.2)
        add.d f16, f0, f14    ; (4.2)
        s.d f4,0(r2)          ; (1.3)
        s.d f8,8(r2)          ; (2.3)
        s.d f12,16(r2)        ; (3.3)
        s.d f16,24(r2)        ; (4.3)
        dadd r1, r1, #32
        dadd r2, r2, #32
        dsub r4, r3, r1
        bnez r4, loop
```

Loop code replicated 4 times and *optimized:* (2/2)

⇒ Dependent instructions have been separated by inserting as many instructions between them as the number of required stalls in the worst case (3 cycles between add.d and s.d).

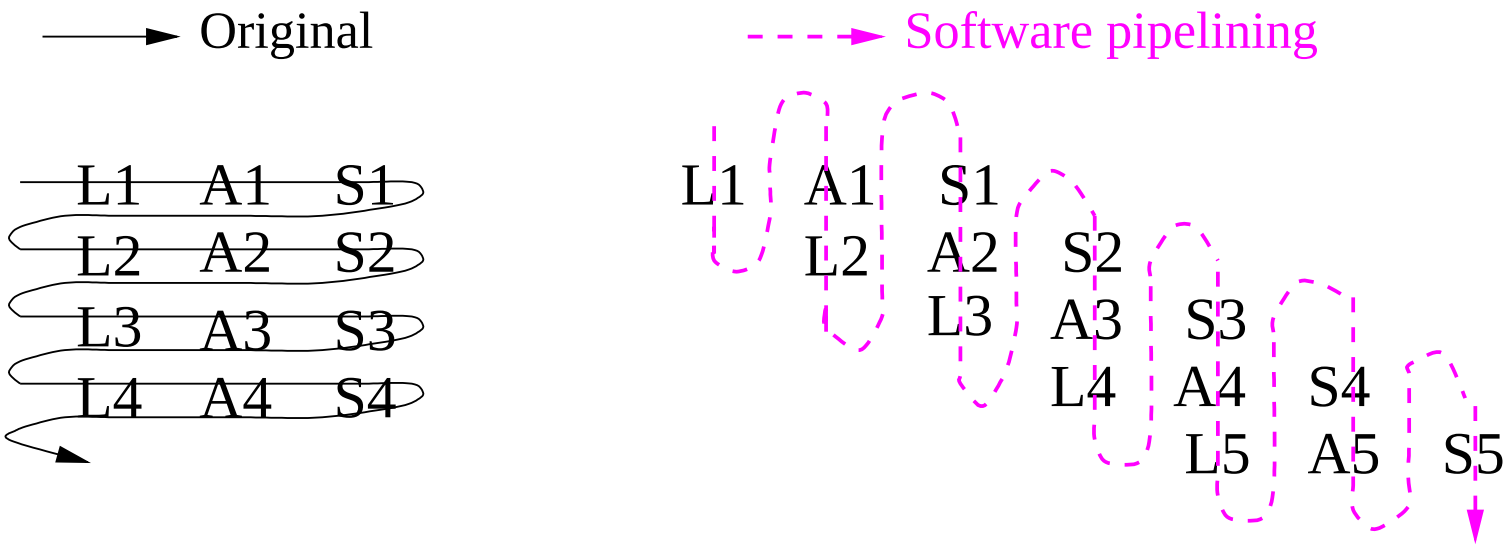
`l.d f2,0(r1)`	IF	ID	L1	L2	WB													
`l.d f6,8(r1)`		IF	ID	L1	L2	WB												
`l.d f10,16(r1)`			IF	ID	L1	L2	WB											
`l.d f14,24(r1)`				IF	ID	L1	L2	WB										
`add.d f4,f0,f2`					IF	ID	A1	A2	A3	A4	WB							
`add.d f8,f0,f6`						IF	ID	A1	A2	A3	A4	WB						
`add.d f12,f0,f10`							IF	ID	A1	A2	A3	A4	WB					
`add.d f16,f0,f14`								IF	ID	A1	A2	A3	A4	WB				
`s.d f4,0(r2)`									IF	ID	S1	S2						
`s.d f8,8(r2)`										IF	ID	S1	S2					
`s.d f12,16(r2)`											IF	ID	S1	S2				
`s.d f16,24(r2)`												IF	ID	S1	S2			

⇒ No stall is required.

Contents

Basic idea (1/2)

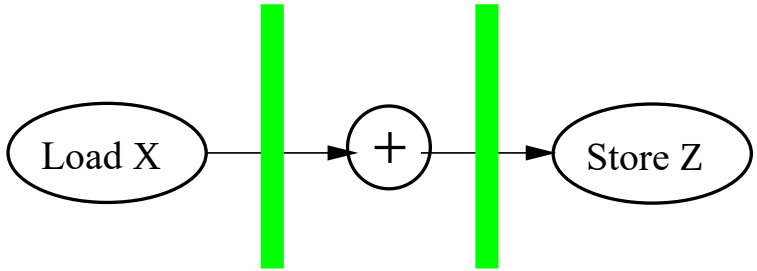
Transform a loop with dependent instructions and independent iterations into another loop with dependent iterations and independent instructions:



Basic idea (2/2)

The name of the technique relates to its ability to process the original loop by simulating the behavior of a pipelined unit:

- The “data” of the pipelined unit are the iterations.
- To avoid intermediate results rewriting, execution is performed from the last stage to the first one.




```
...
lt. i     l.d f2,0(r1)   add.d f4,f0,f2   s.d f4,0(r2)
lt. i+1                  l.d f2,8(r1)     add.d f4,f0,f2   s.d f4,8(r2)
lt. i+2                                   l.d f2,16(r1)    add.d f4,f0,f2   s.d f4,16(r2)
...
```

Code with software pipelining (1/2)

```
start:
    daddi r1, r0, y      ; r1 = y address
    daddi r2, r0, z      ; r2 = z address
    l.d f0, a(r0)       ; f0 = a
    daddi r3, r1, #512 ; 64 elements are 512 bytes

preparation:
    l.d f2, 0(r1)      ; Reads it. 0
    add.d f4, f0, f2 ; Computes it. 0
    l.d f2, 8(r1)      ; Reads it. 1
    daddi r1 ,r1, #16
```

Code with software pipelining (2/2)

```
loop:
     s.d f4,0(r2)          ; Writes it. i
     add.d f4, f0, f2      ; Computes it. i+1
     l.d f2, 0(r1)         ; Reads it. i+2
     daddi r1, r1, #8
     daddi r2, r2, #8
     dsub r4, r3, r1
     bnez r4, loop

end:
     s.d f4,0(r2)          ; Writes it. n-2
     add.d f4, f0, f2      ; Computes it. n-1
     s.d f4,8(r2)          ; Writes it. n-1
```