Instruction scheduling

Akim Demaille, Etienne Renault, Roland Levillain

May 9, 2016

Table of contents

Dependencies analysis 1/2

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

Dependencies analysis 1/2

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

- The 3 following instructions are independent

$$\begin{aligned} inst_1 &: \quad a \leftarrow 42 \\ inst_2 &: \quad b \leftarrow 51 \\ inst_3 &: \quad c \leftarrow 0 \end{aligned}$$

Dependencies analysis 1/2

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

- The 3 following instructions are independent

$$\begin{array}{ll} inst_1: & a \leftarrow 42 \\ inst_2: & b \leftarrow 51 \\ inst_3: & c \leftarrow 0 \end{array}$$

- $inst_1$, $inst_2$ and *$inst_3$* can then be reordered

$$\begin{array}{ll} inst_1: & a \leftarrow 42 \\ inst_2: & b \leftarrow 51 \\ inst_3: & c \leftarrow 0 \end{array} \Bigg\| \begin{array}{ll} inst_1: & a \leftarrow 42 \\ inst_3: & c \leftarrow 0 \\ inst_2: & b \leftarrow 51 \end{array} \Bigg\| \begin{array}{ll} inst_3: & c \leftarrow 0 \\ inst_1: & a \leftarrow 42 \\ inst_2: & b \leftarrow 51 \end{array}$$

$$\begin{array}{ll} inst_1: & c \leftarrow 0 \\ inst_2: & b \leftarrow 51 \\ inst_3: & a \leftarrow 42 \end{array} \Bigg\| \begin{array}{ll} inst_1: & b \leftarrow 51 \\ inst_3: & c \leftarrow 0 \\ inst_2: & a \leftarrow 42 \end{array} \Bigg\| \begin{array}{ll} inst_3: & b \leftarrow 51 \\ inst_1: & a \leftarrow 42 \\ inst_2: & c \leftarrow 0 \end{array}$$

Dependencies analysis 2/2

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

Dependencies analysis 2/2

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

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

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

Dependencies analysis 2/2

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

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

$$\begin{aligned} inst_1: &\quad a \leftarrow 42 \\ inst_2: &\quad b \leftarrow a + 51 \\ inst_3: &\quad c \leftarrow b \times 12 \end{aligned}$$

- 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.

Read after Write (RAW)

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

inst_1 : `lw   $2, 0($4)`
inst_2 : `addi $6, $2, 42`

Read after Write (RAW)

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

inst_1 : `lw   $2, 0($4)`
inst_2 : `addi $6, $2, 42`

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

Write after Read (WAR)

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

Write after Read (WAR)

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

$inst_1$: `lw   $2, 0($4)`
$inst_2$: `addi $4, $12, 42`

Write after Read (WAR)

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

inst_1 : `lw   $2, 0($4)`
inst_2 : `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.

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`

Write after Write (WAW)

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

$$\begin{aligned} \text{inst}_1: &\quad \texttt{add \$1, \$2, \$3} \\ \text{inst}_2: &\quad \texttt{add \$1, \$5, \$6} \end{aligned}$$

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

Why and When reordering?

We would like to reorder the instructions within each basic block in a way which:

- preserves the dependencies between those instructions (and hence the correctness of the program)
- achieves the minimum possible number of pipeline stalls, i.e. two instructions simultaneously in the pipeline manipulates same data, registers, etc.

The two problems can be addressed separately (whew!).

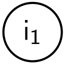
Preserving and computing dependencies?

We construct a directed acyclic graph (DAG) to represent the dependencies between instructions:

- For each instruction in the basic block, create a corresponding vertex in the graph
- For each dependency between two instructions, create a corresponding (annotated) edge in the graph. Note that this edge is annotated.

Computing the dependency graph

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	

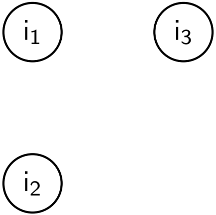


Computing the dependency graph

i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			

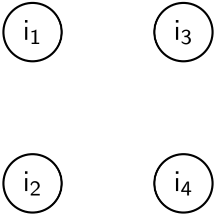
Computing the dependency graph

i_1 : `lw  $1,0($10)`
i_2 : `lw  $2,4($10)`
i_3 : `add $3,$1,$2`
i_4 : `sw  $3,12($10)`
i_5 : `lw  $4,8($10)`
i_6 : `add $3,$1,$4`
i_7 : `sw  $3,16($10)`



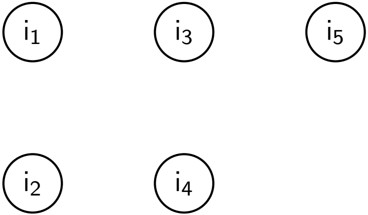
Computing the dependency graph

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



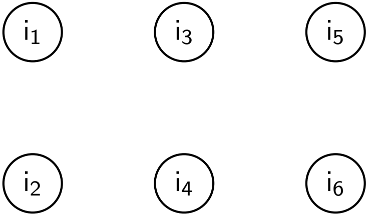
Computing the dependency graph

i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			



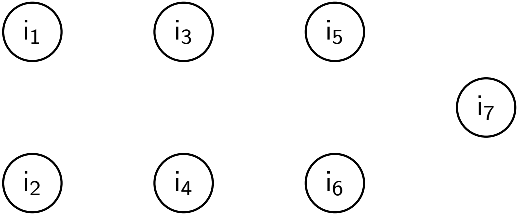
Computing the dependency graph

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



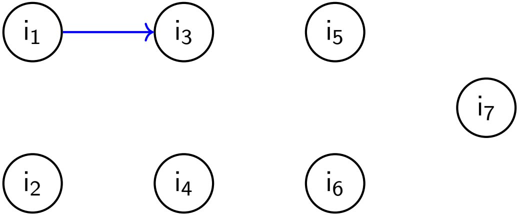
Computing the dependency graph

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



Computing the dependency graph

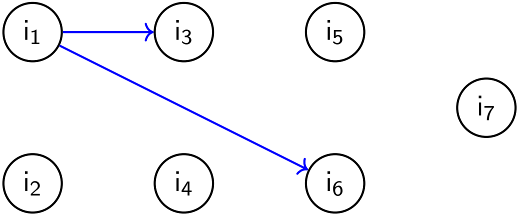
```
i1 :  lw   $1,0($10)  | i4 :  sw   $3,12($10) | i7 :  sw   $3,16($10)
i2 :  lw   $2,4($10)  | i5 :  lw   $4,8($10)  |
i3 :  add  $3,$1,$2   | i6 :  add  $3,$1,$4   |
```



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

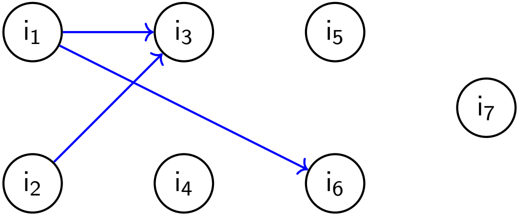
i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

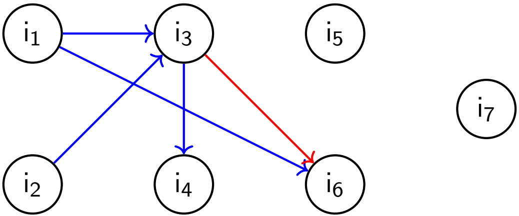
```
i1 : lw  $1,0($10)  | i4 : sw  $3,12($10) | i7 : sw  $3,16($10)
i2 : lw  $2,4($10)  | i5 : lw  $4,8($10)  |
i3 : add $3,$1,$2   | i6 : add $3,$1,$4   |
```



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

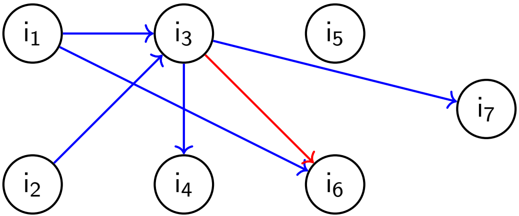
i_1 :	`lw $1,0($10)`	i_4 :	`sw $3,12($10)`	i_7 :	`sw $3,16($10)`
i_2 :	`lw $2,4($10)`	i_5 :	`lw $4,8($10)`		
i_3 :	`add $3,$1,$2`	i_6 :	`add $3,$1,$4`		



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

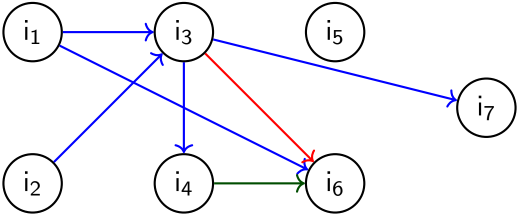
i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

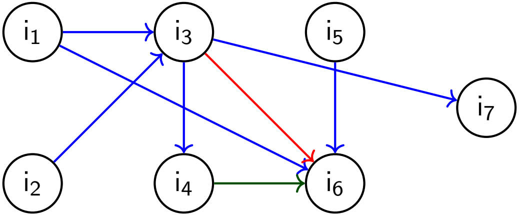
i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

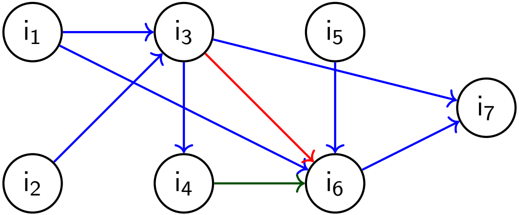
i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

i_1 :	lw	$1,0($10)	i_4 :	sw	$3,12($10)	i_7 :	sw	$3,16($10)
i_2 :	lw	$2,4($10)	i_5 :	lw	$4,8($10)			
i_3 :	add	$3,$1,$2	i_6 :	add	$3,$1,$4			



Type of dependency: RAW, WAW, WAR

Preserving dependencies: Critical Path 1/2

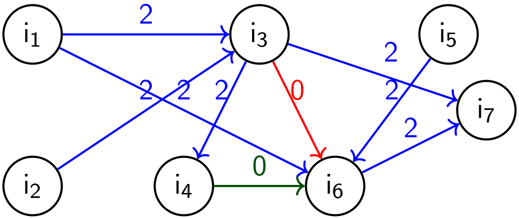
The critical path represents the longest path between two nodes. We add **delays** (weights) to edges:

- 0 for WAW and WAR dependencies
- 2 for RAW dependencies with memory access
- 1 for other RAW dependencies

Preserving dependencies: Critical Path 1/2

The critical path represents the longest path between two nodes. We add **delays** (weights) to edges:

- 0 for WAW and WAR dependencies
- 2 for RAW dependencies with memory access
- 1 for other RAW dependencies



Preserving dependencies: Critical Path 2/2

Any (reverse) topological sort of this DAG (i.e. any linear ordering of the vertices which keeps all the edges pointing forwards) will maintain the dependencies and hence preserve the correctness of the program.

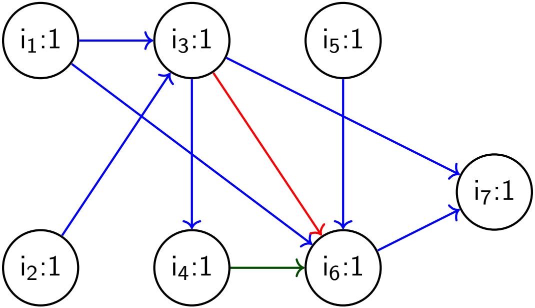
Algorithm:

- Associate a weight 1 to all "instruction node"
- For all nodes n_i in topological postorder
  - If n_i is not a leaf
    - ★ For all nodes n_j in succ(n_i) do
      n_i.weight $\leftarrow$ max (n_i.weight, n_j.weight$+$ delay(n_i, n_j))

Remember "important" edges during computations, they will form the critical path.

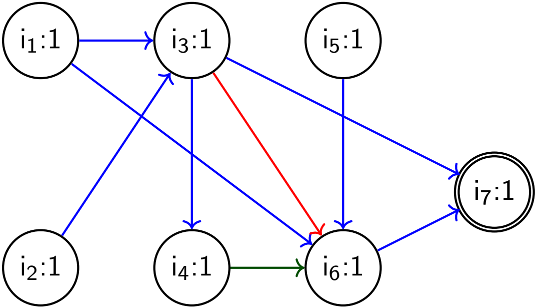
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

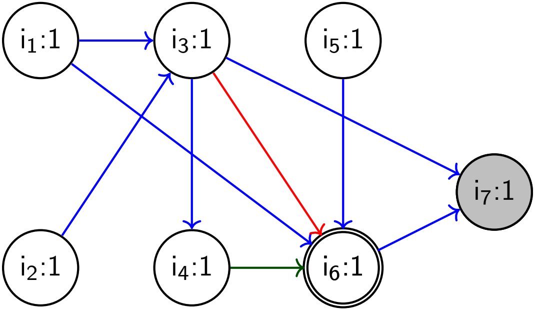
Delays: blue arrows 2, red and green 0



i_7 doesn't have successors, skip it!

Computing the critical path

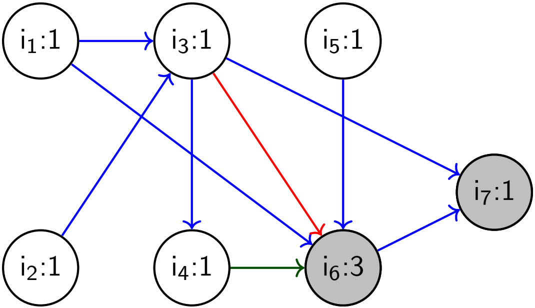
Delays: blue arrows 2, red and green 0



delay(i_6, i_7)=2 > 1, change i_6 weight!

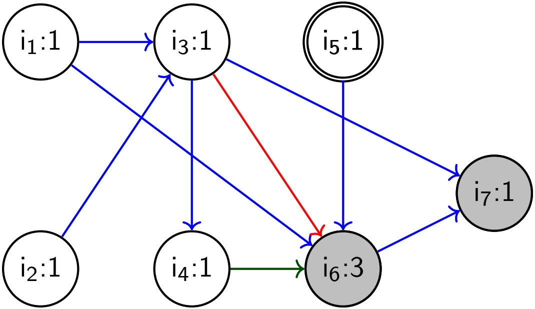
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

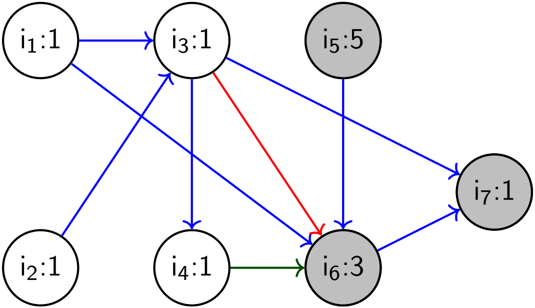
Delays: blue arrows 2, red and green 0



delay(i_5, i_6)$=2 > 1$, change i_5 weight!

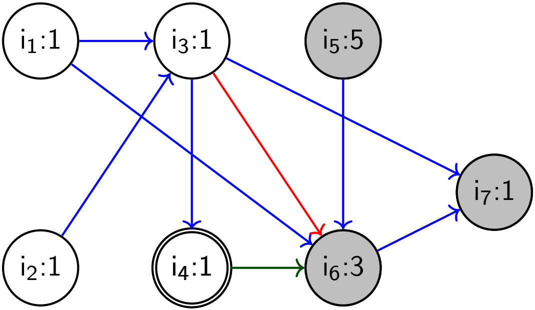
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

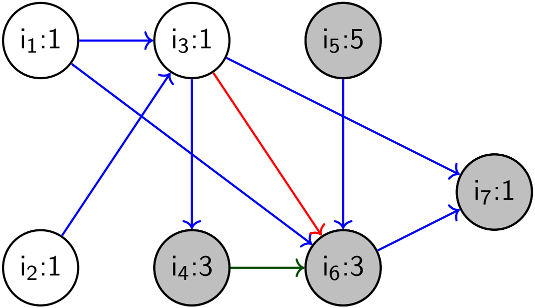
Delays: blue arrows 2, red and green 0



i_6.weight$=3 > 1$, change i_4 weight!

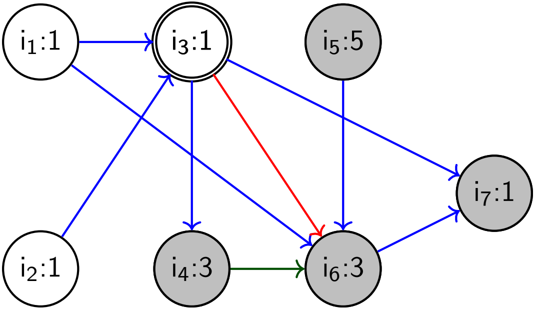
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

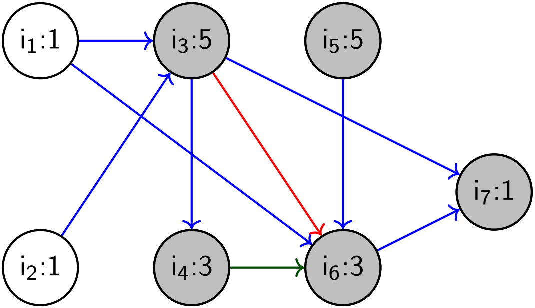
Delays: blue arrows 2, red and green 0



delay(i_3, i_4) + i_4.weight=3 > 1, change i_3 weight!

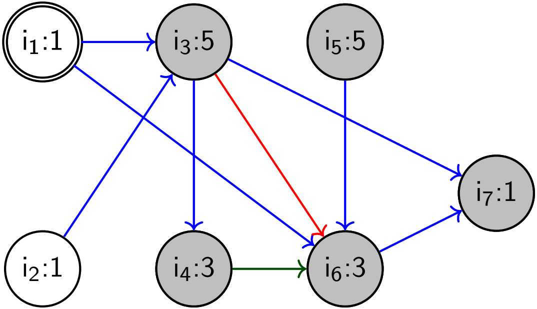
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

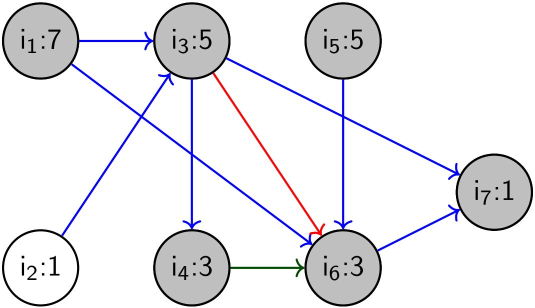
Delays: blue arrows 2, red and green 0



delay(i_1, i_3) + i_3.weight=7 > 1, change i_1 weight!

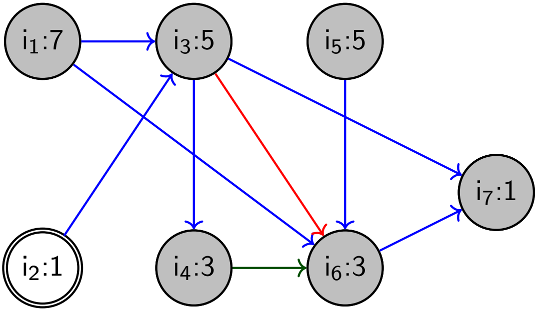
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

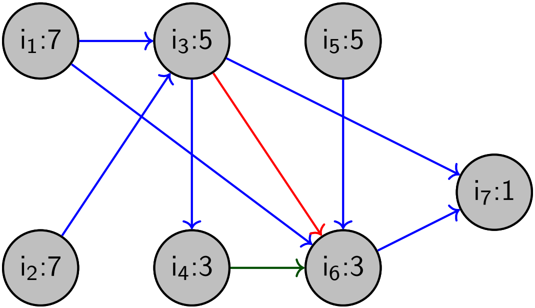
Delays: blue arrows 2, red and green 0



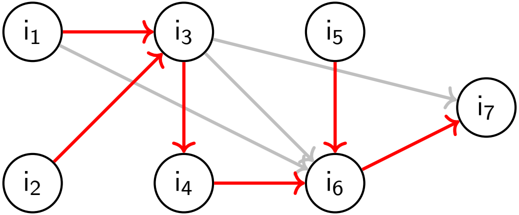
delay(i_2, i_3) + i_3.weight=7 > 1, change i_2 weight!

Computing the critical path

Delays: blue arrows 2, red and green 0



So many orders . . . with one critial path

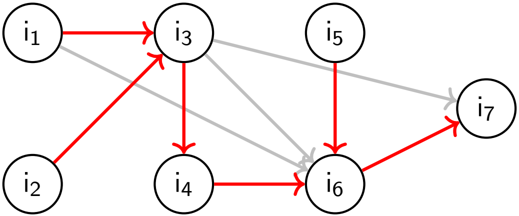


$i_1, i_2, i_3, i_4, i_5, i_6, i_7$ $i_1, i_2, i_3, i_5, i_4, i_6, i_7$ $i_2, i_1, i_3, i_5, i_4, i_6, i_7$ $i_2, i_1, i_3, i_4, i_5, i_6, i_7$

$i_1, i_2, i_5, i_3, i_4, i_6, i_7$ $i_2, i_1, i_5, i_3, i_4, i_6, i_7$ $i_1, i_5, i_2, i_3, i_4, i_6, i_7$ $i_2, i_5, i_1, i_3, i_4, i_6, i_7$

$i_5, i_1, i_2, i_3, i_4, i_6, i_7$ $i_5, i_2, i_1, i_3, i_4, i_6, i_7$

So many orders . . . with one critial path



$i_1,i_2,i_3,i_4,i_5,i_6,i_7$ $i_1,i_2,i_3,i_5,i_4,i_6,i_7$ $i_2,i_1,i_3,i_5,i_4,i_6,i_7$ $i_2,i_1,i_3,i_4,i_5,i_6,i_7$

$i_1,i_2,i_5,i_3,i_4,i_6,i_7$ $i_2,i_1,i_5,i_3,i_4,i_6,i_7$ $i_1,i_5,i_2,i_3,i_4,i_6,i_7$ $i_2,i_5,i_1,i_3,i_4,i_6,i_7$

$i_5,i_1,i_2,i_3,i_4,i_6,i_7$ $i_5,i_2,i_1,i_3,i_4,i_6,i_7$

All these permutations respect dependencies
but is there a best instruction scheduling?

Performances and Pipeline

Not all orders are equivalents!

- Some dependencies can bring hazards that slow down performances inside of the pipeline
- Hazard occurs when:
  - 1 instruction requires the previous instruction has finished
  - 2 instructions need the same data at the same time: one of the two is blocked

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.

	cycle_1	cycle_2	cycle_3	cycle_4	cycle_5	cycle_6	cycle_7	cycle_8	cycle_9
instr_1	IF	ID	EX	ME	WB				
instr_2		IF	ID	EX	ME	WB			
instr_3			IF	ID	EX	ME	WB		
instr_4				IF	ID	EX	ME	WB	
instr_5					IF	ID	EX	ME	WB

Hazard: RAW dependencies 1/2

Some instruction requires a result computed by a previous one!

Consider the following example:

	$cycle_1$	$cycle_2$	$cycle_3$	$cycle_4$	$cycle_5$	$cycle_6$	$cycle_7$
lw $2, 0($4)	IF	ID	EX	ME	WB		
addi $5, $2, 10		IF	ID		EX	ME	WB

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

The goal of `risc` architectures is to produce one per cycle!

Hazard: RAW dependencies 2/2

Consider now the following example:

	cycle$_1$	cycle$_2$	cycle$_3$	cycle$_4$	cycle$_5$	cycle$_6$	cycle$_7$	cycle$_8$
lw $2, 0($4)	IF	ID	EX	ME	WB			
addi $5, $2, 10		IF	ID		EX	ME	WB	
add $12, $9, $11			IF		ID	EX	ME	WB

Hazard: RAW dependencies 2/2

Consider now the following example:

	$cycle_1$	$cycle_2$	$cycle_3$	$cycle_4$	$cycle_5$	$cycle_6$	$cycle_7$	$cycle_8$
lw $2, 0($4)	IF	ID	EX	ME	WB			
addi $5, $2, 10		IF	ID		EX	ME	WB	
add $12, $9, $11			IF		ID	EX	ME	WB

Let's look . . . instruction 3 is independent from the others

Hazard: RAW dependencies 2/2

Consider now the following example:

	$cycle_1$	$cycle_2$	$cycle_3$	$cycle_4$	$cycle_5$	$cycle_6$	$cycle_7$	$cycle_8$
lw \$2, 0(\$4)	IF	ID	EX	ME	WB			
addi \$5, \$2, 10		IF	ID		EX	ME	WB	
add \$12, \$9, \$11			IF		ID	EX	ME	WB

Let's look . . . instruction 3 is independent from the others so we can change the order!

	$cycle_1$	$cycle_2$	$cycle_3$	$cycle_4$	$cycle_5$	$cycle_6$	$cycle_7$	$cycle_8$
lw \$2, 0(\$4)	IF	ID	EX	ME	WB			
add \$12, \$9, \$11		IF	ID	EX	ME	WB		
addi \$5, \$2, 10			IF	ID	EX	ME	WB	

Hazard: WAW dependencies

Two instructions write in the same register!

Consider the following example:

	$cycle_1$	$cycle_2$	$cycle_3$	$cycle_4$	$cycle_5$	$cycle_6$
addi $5, $11, 42	IF	ID	EX	ME	WB	
addi $5, $2, 10		IF	ID	EX	ME	WB

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:

	$cycle_1$	$cycle_2$	$cycle_3$	$cycle_4$	$cycle_5$	$cycle_6$
addi \$5, \$11, 42	IF	ID	EX	ME	WB	
addi \$11, \$2, 10		IF	ID	EX	ME	WB

WAR do not produce stalls !

Back to the example – without scheduling

i_1 :	`lw $1,0($10)`	i_4 :	`sw $3,12($10)`	i_7 :	`sw $3,16($10)`
i_2 :	`lw $2,4($10)`	i_5 :	`lw $4,8($10)`		
i_3 :	`add $3,$1,$2`	i_6 :	`add $3,$1,$4`		

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}
i_1	IF	ID	EX	ME	WB								
i_2		IF	ID	EX	ME	WB							
i_3			IF	ID		EX	ME	WB					
i_4				IF		ID	EX	ME	WB				
i_5						IF	ID	EX	ME	WB			
i_6							IF	ID		EX	ME	WB	
i_7								IF		ID	EX	ME	WB

Back to the example – without scheduling

i_1 :	`lw $1,0($10)`		i_4 :	`sw $3,12($10)`		i_7 :	`sw $3,16($10)`
i_2 :	`lw $2,4($10)`		i_5 :	`lw $4,8($10)`			
i_3 :	`add $3,$1,$2`		i_6 :	`add $3,$1,$4`			

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}
i_1	IF	ID	EX	ME	WB								
i_2		IF	ID	EX	ME	WB							
i_3			IF	ID		EX	ME	WB					
i_4				IF		ID	EX	ME	WB				
i_5						IF	ID	EX	ME	WB			
i_6							IF	ID		EX	ME	WB	
i_7								IF		ID	EX	ME	WB

Without scheduling: 2 dependencies, 2 stalls, 13 cycles!

Minimizing Stalls – First approach

Each time we emit the next instruction, we should try to choose one which

- P_1 does not conflict with the previous emitted instruction
- P_2: is most likely to conflict if first of a pair (e.g. prefer `lw` to `add`)
- P_3: is as far away as possible (along paths in the DAG) from an instruction which can validly be scheduled last

Minimizing Stalls – First approach

Each time we emit the next instruction, we should try to choose one which

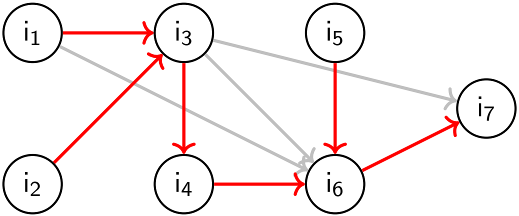
- P_1 does not conflict with the previous emitted instruction
- P_2: is most likely to conflict if first of a pair (e.g. prefer `lw` to `add`)
- P_3: is as far away as possible (along paths in the DAG) from an instruction which can validly be scheduled last

Algorithm:

- Compute the dependency graph
- While the list of candidate instructions is not empty
  - If one instruction satisfies P_1, P_2, and P_3: remove it from the list and emit it.
    - Remove the instruction from the DAG and insert the newly minimal elements into the candidate list.
  - Otherwise emit a `nop` instruction

Applying scheduling algorithm to the example

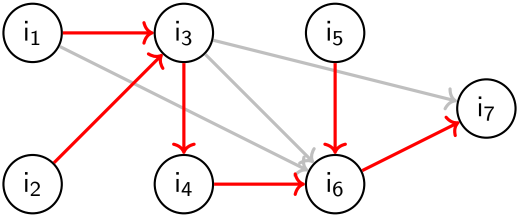
i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			



$$\text{Candidates} = \{i_1, i_2, i_5\}$$
$$\text{Final Order} =$$

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



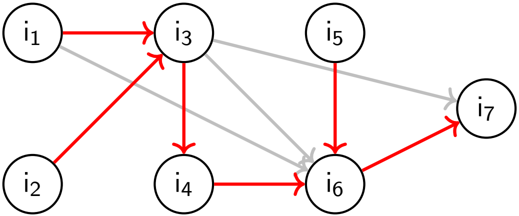
Candidates $= \{i_1, i_2, i_5\}$

Final Order $=$

Choose i_1 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 :	lw	\$1,0(\$10)	i_4 :	sw	\$3,12(\$10)	i_7 :	sw	\$3,16(\$10)
i_2 :	lw	\$2,4(\$10)	i_5 :	lw	\$4,8(\$10)			
i_3 :	add	\$3,\$1,\$2	i_6 :	add	\$3,\$1,\$4			



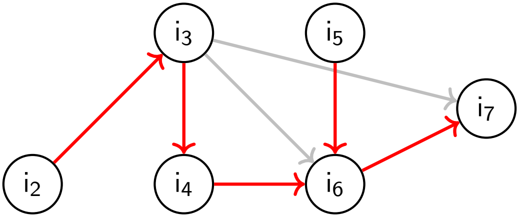
Candidates = $\{i_1, i_2, i_5\}$

Final Order = i_1

Choose i_1 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



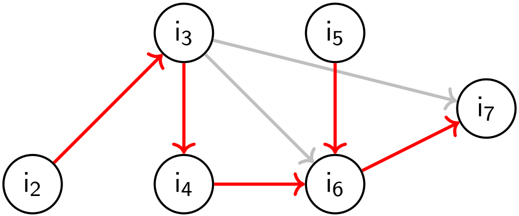
Candidates $= \{i_1, i_2, i_5\}$

Final Order $= i_1$

Choose i_1 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

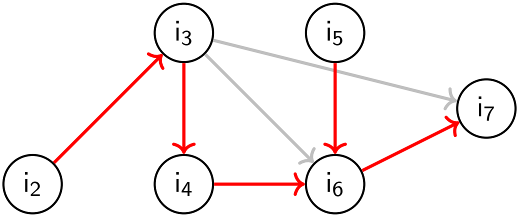
i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



$$\text{Candidates} = \{i_2, i_5\}$$
$$\text{Final Order} = i_1$$

Applying scheduling algorithm to the example

i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			

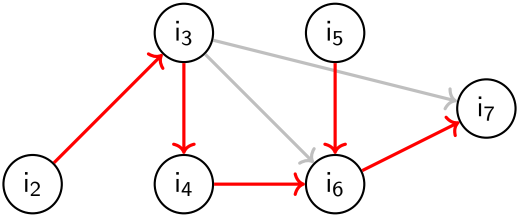


$$\text{Candidates} = \{i_2, i_5\}$$
$$\text{Final Order} = i_1$$

Choose i_2 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

```
i1 :  lw  $1,0($10)  | i4 :  sw  $3,12($10) | i7 :  sw  $3,16($10)
i2 :  lw  $2,4($10)  | i5 :  lw  $4,8($10)  |
i3 :  add $3,$1,$2   | i6 :  add $3,$1,$4   |
```



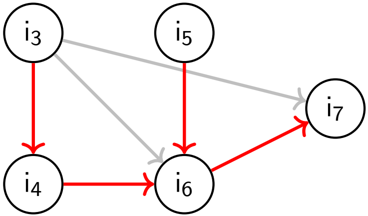
Candidates = $\{i_2, i_5\}$

Final Order = i_1, i_2

Choose i_2 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			

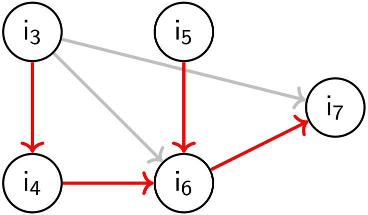


$$\text{Candidates} = \{i_2, i_5\}$$
$$\text{Final Order} = i_1, i_2$$

Choose i_2 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

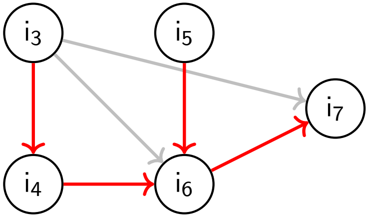
i_1 :	`lw $1,0($10)`	i_4 :	`sw $3,12($10)`	i_7 :	`sw $3,16($10)`
i_2 :	`lw $2,4($10)`	i_5 :	`lw $4,8($10)`		
i_3 :	`add $3,$1,$2`	i_6 :	`add $3,$1,$4`		



$$\text{Candidates} = \{i_5, i_3\}$$
$$\text{Final Order} = i_1, i_2$$

Applying scheduling algorithm to the example

i_1 :	`lw $1,0($10)`	i_4 :	`sw $3,12($10)`	i_7 :	`sw $3,16($10)`
i_2 :	`lw $2,4($10)`	i_5 :	`lw $4,8($10)`		
i_3 :	`add $3,$1,$2`	i_6 :	`add $3,$1,$4`		

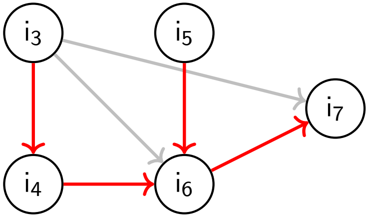


$$\begin{aligned} \text{Candidates} &= \{i_5, i_3\} \\ \text{Final Order} &= i_1, i_2 \end{aligned}$$

Choose i_5 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 :	lw	$1,0($10)	i_4 :	sw	$3,12($10)	i_7 :	sw	$3,16($10)
i_2 :	lw	$2,4($10)	i_5 :	lw	$4,8($10)			
i_3 :	add	$3,$1,$2	i_6 :	add	$3,$1,$4			

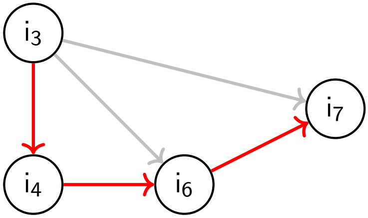


$$\text{Candidates} = \{i_5, i_3\}$$
$$\text{Final Order} = i_1, i_2, i_5$$

Choose i_5 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			



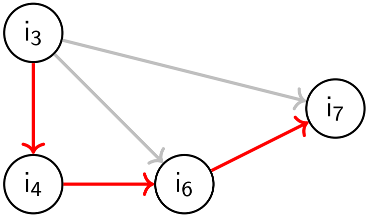
Candidates $= \{i_5, i_3\}$

Final Order $= i_1, i_2, i_5$

Choose i_5 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

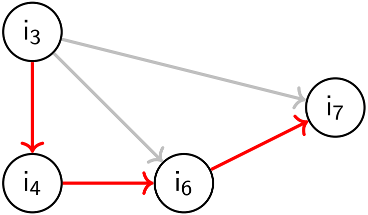
i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			



$$\text{Candidates} = \{i_3\}$$
$$\text{Final Order} = i_1, i_2, i_5$$

Applying scheduling algorithm to the example

i_1 :	lw	$1,0($10)	i_4 :	sw	$3,12($10)	i_7 :	sw	$3,16($10)
i_2 :	lw	$2,4($10)	i_5 :	lw	$4,8($10)			
i_3 :	add	$3,$1,$2	i_6 :	add	$3,$1,$4			

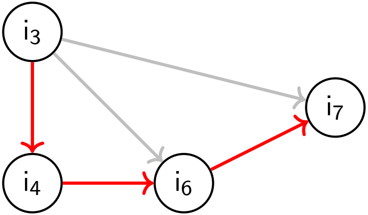


$$\text{Candidates} = \{i_3\}$$
$$\text{Final Order} = i_1, i_2, i_5$$

Choose i_3 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	

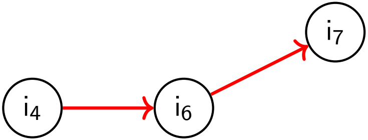


$$\begin{aligned} \text{Candidates} &= \{i_3\} \\ \text{Final Order} &= i_1, i_2, i_5, i_3 \end{aligned}$$

Choose i_3 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			

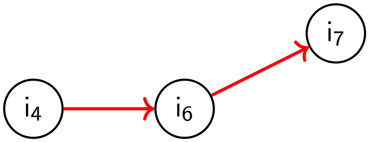


$$\begin{aligned} \text{Candidates} &= \{i_3\} \\ \text{Final Order} &= i_1, i_2, i_5, i_3 \end{aligned}$$

Choose i_3 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

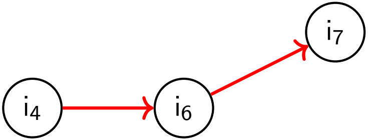
i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			



$$\text{Candidates} = \{i_4\}$$
$$\text{Final Order} = i_1, i_2, i_5, i_3$$

Applying scheduling algorithm to the example

i_1 :	lw	$1,0($10)	i_4 :	sw	$3,12($10)	i_7 :	sw	$3,16($10)
i_2 :	lw	$2,4($10)	i_5 :	lw	$4,8($10)			
i_3 :	add	$3,$1,$2	i_6 :	add	$3,$1,$4			

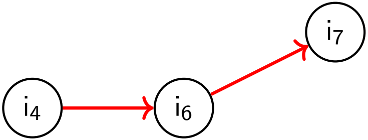


$$\begin{aligned} \text{Candidates} &= \{i_4\} \\ \text{Final Order} &= i_1, i_2, i_5, i_3 \end{aligned}$$

Choose i_4 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	

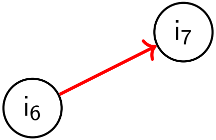


$$\text{Candidates} = \{i_4\}$$
$$\text{Final Order} = i_1, i_2, i_5, i_3, i_4$$

Choose i_4 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 :	lw	$1,0($10)	i_4 :	sw	$3,12($10)	i_7 :	sw	$3,16($10)
i_2 :	lw	$2,4($10)	i_5 :	lw	$4,8($10)			
i_3 :	add	$3,$1,$2	i_6 :	add	$3,$1,$4			

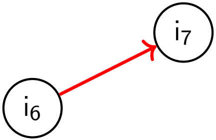


$$\text{Candidates} = \{i_4\}$$
$$\text{Final Order} = i_1, i_2, i_5, i_3, i_4$$

Choose i_4 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

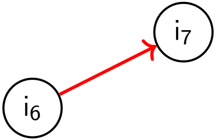
i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



$$\text{Candidates} = \{i_6\}$$
$$\text{Final Order} = i_1, i_2, i_5, i_3, i_4$$

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	

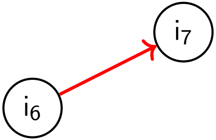


$$\begin{aligned}\text{Candidates} &= \{i_6\} \\ \text{Final Order} &= i_1, i_2, i_5, i_3, i_4\end{aligned}$$

Choose i_6 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			



$$\text{Candidates} = \{i_6\}$$
$$\text{Final Order} = i_1, i_2, i_5, i_3, i_4, i_6$$

Choose i_6 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	

$$\text{Candidates} = \{i_6\}$$
$$\text{Final Order} = i_1, i_2, i_5, i_3, i_4, i_6$$

Choose i_6 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	

$$\text{Candidates} = \{i_7\}$$
$$\text{Final Order} = i_1, i_2, i_5, i_3, i_4, i_6$$

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	

Candidates = $\{i_7\}$

Final Order = $i_1, i_2, i_5, i_3, i_4, i_6$

Choose i_7 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 :	`lw $1,0($10)`	i_4 :	`sw $3,12($10)`	i_7 :	`sw $3,16($10)`
i_2 :	`lw $2,4($10)`	i_5 :	`lw $4,8($10)`		
i_3 :	`add $3,$1,$2`	i_6 :	`add $3,$1,$4`		

Candidates = $\{i_7\}$

Final Order = $i_1, i_2, i_5, i_3, i_4, i_6, i_7$

Choose i_7 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	

$$\begin{aligned} \text{Candidates} &= \{i_7\} \\ \text{Final Order} &= i_1, i_2, i_5, i_3, i_4, i_6, i_7 \end{aligned}$$

Choose i_7 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

```
i1 :  lw   $1,0($10)
i2 :  lw   $2,4($10)
i3 :  add  $3,$1,$2
i4 :  sw   $3,12($10)
i5 :  lw   $4,8($10)
i6 :  add  $3,$1,$4
i7 :  sw   $3,16($10)
```

Final Order = i_1, i_2, i_5, i_3, i_4, i_6, i_7

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}
i_1	IF	ID	EX	ME	WB						
i_2		IF	ID	EX	ME	WB					
i_5			IF	ID	EX	ME	WB				
i_3				IF	ID	EX	ME	WB			
i_4					IF	ID	EX	ME	WB		
i_6						IF	ID	EX	ME	WB	
i_7							IF	ID	EX	ME	WB

Applying scheduling algorithm to the example

```
i1 :  lw   $1,0($10)  | i4 :  sw   $3,12($10) | i7 :  sw   $3,16($10)
i2 :  lw   $2,4($10)  | i5 :  lw   $4,8($10)  |
i3 :  add  $3,$1,$2   | i6 :  add  $3,$1,$4   |
```

Final Order = $i_1, i_2, i_5, i_3, i_4, i_6, i_7$

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}
i_1	IF	ID	EX	ME	WB						
i_2		IF	ID	EX	ME	WB					
i_5			IF	ID	EX	ME	WB				
i_3				IF	ID	EX	ME	WB			
i_4					IF	ID	EX	ME	WB		
i_6						IF	ID	EX	ME	WB	
i_7							IF	ID	EX	ME	WB

With scheduling: still 2 dependencies but 0 stalls and 11 cycles!

A word on scheduling strategies

- Sometimes we cannot avoid some stalls
- Computing the critical path can be smarter:
  - Rather than attributing 1 as weight to every instruction, we can adjust according to the real time of executing the instruction
  - We can take advantages of the number of successors
  - ... many yet-to-be-define heuristics!
- Computing the DAG of dependencies can be done in $O(n^2)$ by scanning backwards through the basic block and adding edges as dependencies arise

A word on performances

We can statically compute instructions per cycle IPC$=\frac{\text{nb instructions}}{\text{nb cycles}}$, to evaluate 2 possible scheduling.

In the previous example:

- without scheduling IPC$=\frac{7}{13} = 0.53$
- with scheduling IPC$=\frac{7}{11} = 0.63$ (better!)

We can also statically compute cycle per instructions: CPI $= \frac{1}{\text{IPC}}$.
The CPI lower bound is $\frac{\sum \alpha \times \beta}{\text{nb instructions}}$, avec α is the number of instructions for a given instruction type and β the associated cost.

Can we do better?

Consider the following code (representing a basic block):

```
i1:  Loop:  lw     $t0, 0($s1)      # t0=array element
i2:         addu   $t0, $t0, $s2    # add scalar in s2
i3:         sw     $t0, 0($s1)      # store result
i4:         addi   $s1, $s1,-4      # decrement pointer
i5:         bne    $s1, $0, Loop    # branch s1!=0
```

Can we do better?

Consider the following code (representing a basic block):

```
i1:  Loop:  lw     $t0, 0($s1)        # t0=array element
i2:         addu   $t0, $t0, $s2      # add scalar in s2
i3:         sw     $t0, 0($s1)        # store result
i4:         addi   $s1, $s1,-4        # decrement pointer
i5:         bne    $s1, $0, Loop      # branch s1!=0
```

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}
i_1	IF	ID	EX	ME	WB											
i_2		IF	ID		EX	ME	WB									
i_3			IF					ID	EX	ME	WB					
i_4								IF	ID	EX	ME	WB				
i_5									IF				ID	EX	ME	WB

Can we do better?

Consider the following code (representing a basic block):

```
i1:  Loop:  lw     $t0, 0($s1)       # t0=array element
i2:         addu   $t0, $t0, $s2     # add scalar in s2
i3:         sw     $t0, 0($s1)       # store result
i4:         addi   $s1, $s1,-4       # decrement pointer
i5:         bne    $s1, $0, Loop     # branch s1!=0
```

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}
i_1	IF	ID	EX	ME	WB											
i_2		IF	ID		EX	ME	WB									
i_3			IF					ID	EX	ME	WB					
i_4								IF	ID	EX	ME	WB				
i_5									IF				ID	EX	ME	WB

16 cycles for 5 instructions that are all dependent!

IPC = 0.31

Loop Unrolling

- Replicate loop body to expose more parallelism
- Reduces loop-control overhead

Loop Unrolling

- Replicate loop body to expose more parallelism
- Reduces loop-control overhead

At high level, it can be seen as following:

Without Loop Unrolling	With Loop Unrolling
int i;	int i;
for (i = 0; i < 100; ++i)	for (i = 0; i < 100; i+=5)
tab[i] = tab[i] +42;	tab[i] = tab[i] +42;
	tab[i+1] = tab[i+1] +42;
	tab[i+2] = tab[i+2] +42;
	tab[i+3] = tab[i+3] +42;
	tab[i+4] = tab[i+4] +42;

Loop Unrolling

- Replicate loop body to expose more parallelism
- Reduces loop-control overhead

At high level, it can be seen as following:

Without Loop Unrolling	With Loop Unrolling
int i;	int i;
for (i = 0; i < 100; ++i)	for (i = 0; i < 100; i+=5)
tab[i] = tab[i] +42;	tab[i] = tab[i] +42;
	tab[i+1] = tab[i+1] +42;
	tab[i+2] = tab[i+2] +42;
	tab[i+3] = tab[i+3] +42;
	tab[i+4] = tab[i+4] +42;

Special care must be taken for pre and post loops operations (as well as intra-loop dependencies)

Loop Unrolling – back to the example

i_1:	Loop:	lw	$t0, 0($s1)	# t0=array element
i_2:		addu	$t0, $t0, $s2	# add scalar in s2
i_3:		sw	$t0, 0($s1)	# store result
i_4:		addi	$s1, $s1,-4	# decrement pointer
i_5:		bne	$s1, $0, Loop	# branch s1!=0
i_6:	Loop:	lw	$t0, 0($s1)	# t0=array element
i_7:		addu	$t0, $t0, $s2	# add scalar in s2
i_8:		sw	$t0, 0($s1)	# store result
i_9:		addi	$s1, $s1,-4	# decrement pointer
i_{10}:		bne	$s1, $0, Loop	# branch s1!=0
i_{11}:	Loop:	lw	$t0, 0($s1)	# t0=array element
i_{12}:		addu	$t0, $t0, $s2	# add scalar in s2
i_{13}:		sw	$t0, 0($s1)	# store result
i_{14}:		addi	$s1, $s1,-4	# decrement pointer
i_{15}:		bne	$s1, $0, Loop	# branch s1!=0

First duplicate N times the the body of the loop!

Loop Unrolling – back to the example

```
i1:   Loop:  lw     $t0, 0($s1)      # t0=array element
i2:          addu   $t0, $t0, $s2    # add scalar in s2
i3:          sw     $t0, 0($s1)      # store result
i4:          addi   $s1, $s1,-4      # decrement pointer
i6:          lw     $t0, 0($s1)      # t0=array element
i7:          addu   $t0, $t0, $s2    # add scalar in s2
i8:          sw     $t0, 0($s1)      # store result
i9:          addi   $s1, $s1,-4      # decrement pointer
i11:         lw     $t0, 0($s1)      # t0=array element
i12:         addu   $t0, $t0, $s2    # add scalar in s2
i13:         sw     $t0, 0($s1)      # store result
i14:         addi   $s1, $s1,-4      # decrement pointer
i15:         bne    $s1, $0, Loop    # branch s1!=0
```

Remove redundant labels and jump
(by supposing that we are able to do it!)

Loop Unrolling – back to the example

```
i1:   Loop:  lw     $t0, 0($s1)      # t0=array element
i2:          addu   $t0, $t0, $s2    # add scalar in s2
i3:          sw     $t0, 0($s1)      # store result
i4:          addi   $s1, $s1,-4      # decrement pointer
i6:          lw     $t1, 0($s1)      # t0=array element
i7:          addu   $t1, $t1, $s2    # add scalar in s2
i8:          sw     $t1, 0($s1)      # store result
i9:          addi   $s1, $s1,-4      # decrement pointer
i11:         lw     $t2, 0($s1)      # t0=array element
i12:         addu   $t2, $t2, $s2    # add scalar in s2
i13:         sw     $t2, 0($s1)      # store result
i14:         addi   $s1, $s1,-4      # decrement pointer
i15:         bne    $s1, $0, Loop    # branch s1!=0
```

Use other temporaries name when possible!

Loop Unrolling – back to the example

```
i4:   Loop:  addi   $s1, $s1,-12     # decrement pointer
i1:          lw     $t0, 0($s1)      # t0=array element
i2:          addu   $t0, $t0, $s2    # add scalar in s2
i3:          sw     $t0, 0($s1)      # store result
i6:          lw     $t1, 4($s1)      # t0=array element
i7:          addu   $t1, $t1, $s2    # add scalar in s2
i8:          sw     $t1, 4($s1)      # store result
i11:         lw     $t2, 8($s1)      # t0=array element
i12:         addu   $t2, $t2, $s2    # add scalar in s2
i13:         sw     $t2, 8($s1)      # store result
i15:         bne    $s1, $0, Loop    # branch s1!=0
```

Grab redundant operation and merge them **carefully**!

Loop Unrolling – back to the example

```
i1:   Loop:  addi   $s1, $s1,-12      # decrement pointer for N=3
i2:          lw     $t0, 0($s1)       # t0=array element
i3:          lw     $t1, 4($s1)       # t1=array element
i4:          lw     $t2, 8($s1)       # t2=array element
i5:          addu   $t0, $t0, $s2     # add scalar in s2
i6:          addu   $t1, $t1, $s2     # add scalar in s2
i7:          addu   $t2, $t2, $s2     # add scalar in s2
i8:          sw     $t0, 0($s1)       # store result
i9:          sw     $t1, 4($s1)       # store result
i10:         sw     $t2, 8($s1)       # store result
i11:         bne    $s1, $0, Loop     # branch s1!=0
```

Schedule the instructions and renumber them (and update comments)!

Pros & Cons

- We avoid a lot of conditional jumps (and many stall hence)
- We require 19 cycles for 11 instructions: IPC=0.57
  (a lot better than the previous 0.31)
- This trick allows to have more independent instructions to insert, and thus, less stalls!
- But we have now a prologue and an epilogue: i.e., two more basic blocks
- Require more temporaries: register allocation will be harder!
- Try it by yourself in gcc `-funroll-loops`

A very last word on Branch Hazards 1/2

- Conditional jumps often introduce delays since we cannot pre-fetch instrcutions
  - Branch Outcome and Branch Target Address are ready at the end of the EX stage (3th stage)
  - Conditional branches are solved when PC is updated at the end of the ME stage (4th stage)
- Can we avoid them?

We only know i_{next} at cycle 5!

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
bne \$1,\$2, loop	IF	ID	EX	ME	WB				
nop		IF	ID	EX	ME	WB			
nop			IF	ID	EX	ME	WB		
nop				IF	ID	EX	ME	WB	
i_{next}					IF	ID	EX	ME	WB

A very last word on Branch Hazards 2/2

- X delayed slot: the X instructions after a branch are systematically executed
- The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches
- We need branch prediction... but nowadays, most of processors do it for us (and use `slt`...)!
- Some architectures have bypass between stages to avoid stalls

Avoid as possible floating points and jumps!

A very last word on Branch Hazards 2/2

- X delayed slot: the X instructions after a branch are systematically executed
- The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches
- We need branch prediction... but nowadays, most of processors do it for us (and use `slt`...)!
- Some architectures have bypass between stages to avoid stalls

Avoid as possible floating points and jumps!

"Do you program in `mips`?" she asked. "`nop`", he said.

Stalls due to caches

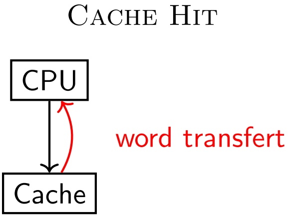
When the processor processor needs to access a data:

- If data is in cache: with a cost of 3 cycles
- Otherwise: with a cost of 100 cycles

Stalls due to caches

When the processor processor needs to access a data:

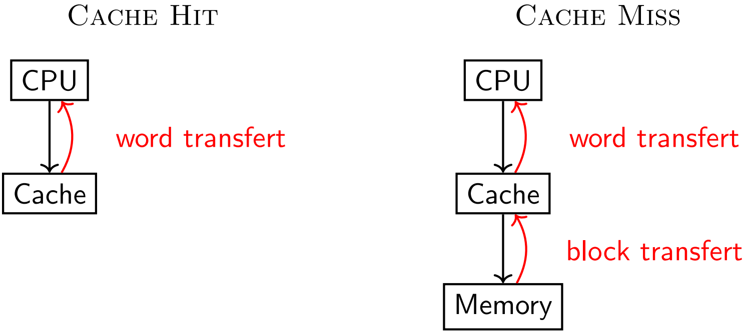
- If data is in cache: with a cost of 3 cycles
- Otherwise: with a cost of 100 cycles



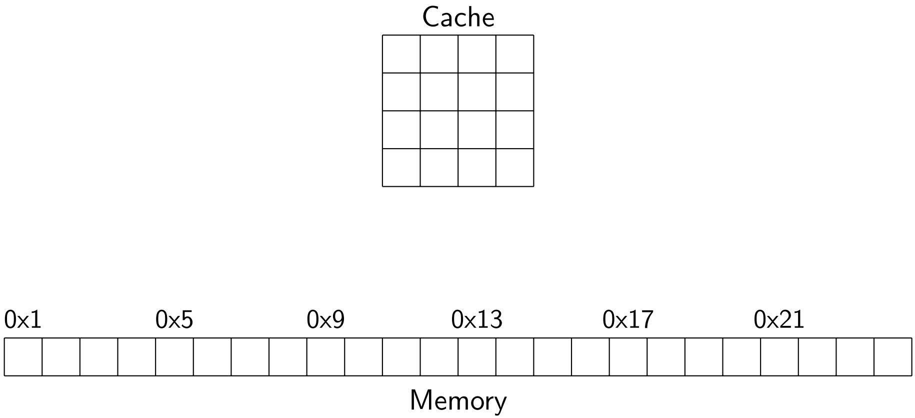
Stalls due to caches

When the processor processor needs to access a data:

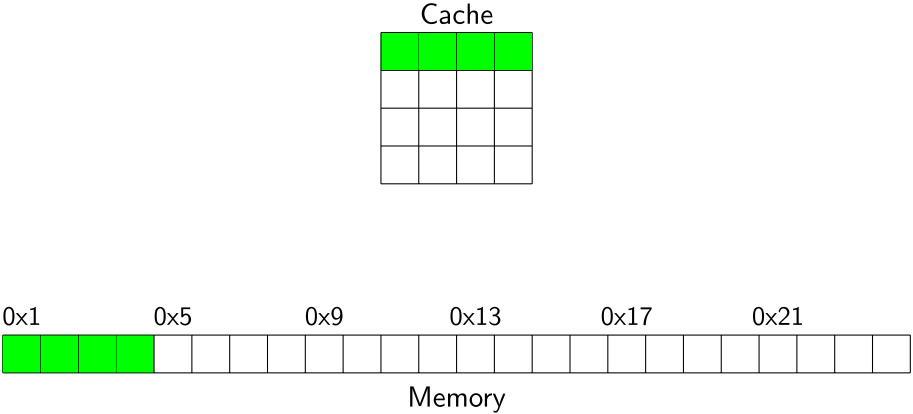
- If data is in cache: with a cost of 3 cycles
- Otherwise: with a cost of 100 cycles



Cache Fundamentals 1/2

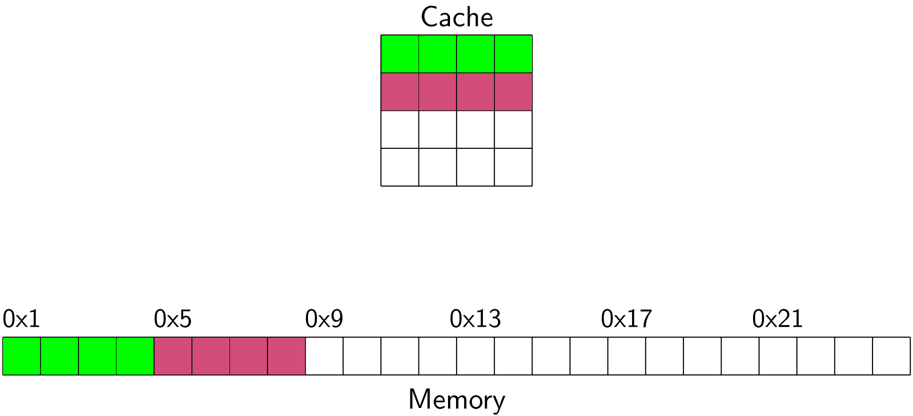


Cache Fundamentals 1/2



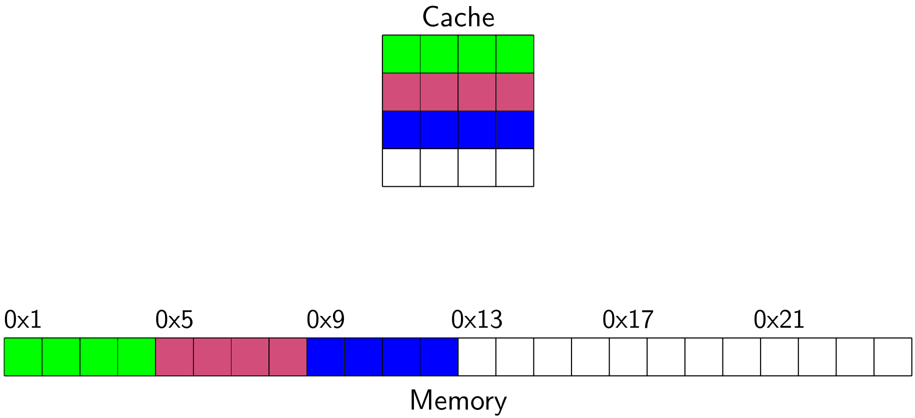
Access to adress 0x1, 4 words are fetched

Cache Fundamentals 1/2



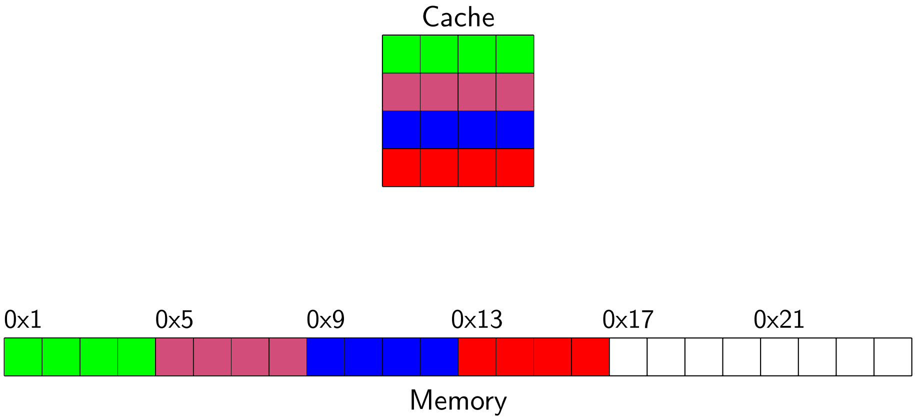
Access to adress 0x5, 4 words are fetched

Cache Fundamentals 1/2



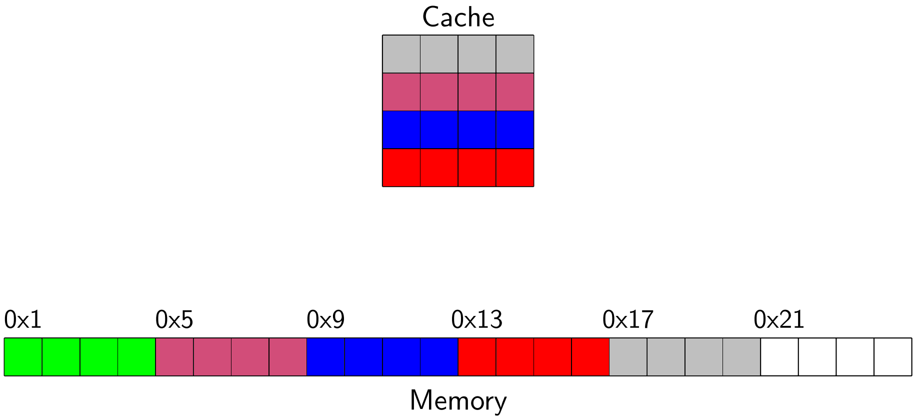
Access to adress 0x9, 4 words are fetched

Cache Fundamentals 1/2



Access to adress 0x13, 4 words are fetched

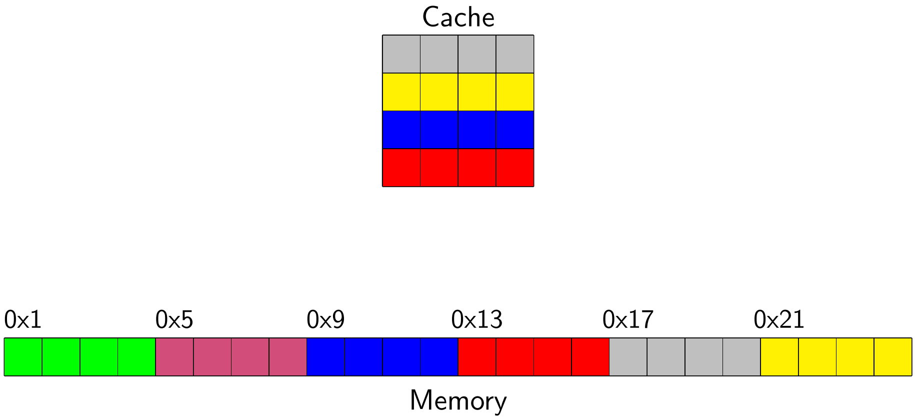
Cache Fundamentals 1/2



Access to adress 0x17, 4 words are fetched

First line of cache is replaced!

Cache Fundamentals 1/2



Access to adress 0x21, 4 words are fetched

Second line of cache is replaced!

Cache Fundamentals 1/2

Many strategies to put data into the cache:

- Direct Mapping:
  - The address is decomposed in 3 parts: `tag` (8b), `line` (22b), and `word`(2b)
  - Each block of main memory maps to only one cache line, i.e. block-size = cache-line-size
  - Simple, Inexpensive, and fixed location for given block
- Associative Mapping:
  - A main memory block can load into any line of cache
  - Memory address is interpreted as tag and word
  - Tag uniquely identifies block of memory
  - Each block of main memory maps to only one cache line, i.e. block-size = cache-line-size
  - Complex, Expensive, and no-fixed location for given block

Prefetching

Fetch the data before it is needed (i.e. pre-fetch) by the program

- Eliminate cache misses
- Involves predicting which address will be needed in the future (as for branch prediction)
- In contrast to branch prediction:
  - incorrect prefetched data will simply not be used
  - there is no need for state recovery

Locality

- Locality is the principle that future memory accesses are near past accesses
- Memories take advantage of two types of locality
  - Temporal locality, i.e. near in time: we will often access the same data again very soon
  - Spatial locality, i.e. near in space/distance: our next access is often very close to our last access (or recent accesses)

Locality

- Locality is the principle that future memory accesses are near past accesses
- Memories take advantage of two types of locality
  - Temporal locality, i.e. near in time: we will often access the same data again very soon
  - Spatial locality, i.e. near in space/distance: our next access is often very close to our last access (or recent accesses)

Some Instruction Set Architecture (ISA) allows to pre-fetch some data: i.e., Humans or compilers has to insert (take advantage) of these instructions

Loops optimisations

We have already seen loops-unrolling to avoid stalls inside of the processor. Other techniques exist to avoid stalls due to cache:

- Loop Fission
- Loop interchanging
- Tabular Grouping
- Loop blocking
- Loop reversal
- Loop tiling
- ...

Loop Fission 1/2

Consider the following code, and direct mapping strategy:

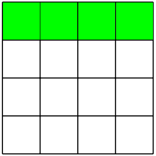
```
int A[1024]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i) {
   A[i] = B[i];
   C[i] = C[i-1] + 1;
}
```

Loop Fission 1/2

Consider the following code, and direct mapping strategy:

```
int A[1024]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i) {
   A[i] = B[i];
   C[i] = C[i-1] + 1;
}
```

Fetch A[i], A[$i+1$], A[$i+2$] and A[$i+3$]

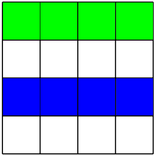


Loop Fission 1/2

Consider the following code, and direct mapping strategy:

```
int A[1024]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i) {
  A[i] = B[i];
  C[i] = C[i-1] + 1;
}
```

Fetch B[i], B[$i+1$], B[$i+2$] and B[$i+3$]

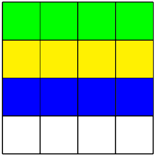


Loop Fission 1/2

Consider the following code, and direct mapping strategy:

```
int A[1024]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i) {
   A[i] = B[i];
   C[i] = C[i-1] + 1;
}
```

Fetch C[i], C[$i+1$], C[$i+2$] and C[$i+3$]

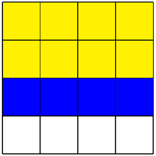


Loop Fission 1/2

Consider the following code, and direct mapping strategy:

```
int A[1024]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i) {
   A[i] = B[i];
   C[i] = C[i-1] + 1;
}
```

Fetch C[$i-1$] will probably conflict



- Hopefully A[i], B[i] and C[i] will not conflict in the cache
- but ... C[i-1] will probably!

Loop Fission 2/2

Solution

Divide the loop into two:

- Less pressure on cache
- We can now insert padding to avoid conflicts

```
int A[1024]; padding[xx]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i)
  A[i] = B[i];
for (int i = 1; i<1024; ++i)
  C[i] = C[i-1] + 1;
```

Try it by yourself in gcc -ftree-loop-distribution

Loop interchanging 1/2

Consider the following code, and direct mapping cache:

```
int A[1024][1024];
for (int j = 1; j<1024; ++j)
   for (int i = 1; i<1024; ++i)
      A[j][i] = A[j][i] * 42;
```

In Fortran, the elements of an array are stored in memory contiguously by column, and the original loop iterates over rows, potentially creating at each access a cache miss

A	B	C
D	E	F

is stored

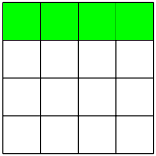
A	D	B	E	C	F

Loop interchanging 1/2

Consider the following code, and direct mapping cache:

```
int A[1024][1024];
for (int j = 1; j<1024; ++j)
  for (int i = 1; i<1024; ++i)
    A[j][i] = A[j][i] * 42;
```

Fetch A[j][i], A[$j+1$][i], A[$j+2$][i], and A[$j+3$][i]



In Fortran, the elements of an array are stored in memory contiguously by column, and the original loop iterates over rows, potentially creating at each access a cache miss

A	B	C
D	E	F

is stored

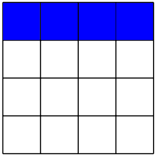
A	D	B	E	C	F

Loop interchanging 1/2

Consider the following code, and direct mapping cache:

```
int A[1024][1024];
for (int j = 1; j<1024; ++j)
  for (int i = 1; i<1024; ++i)
    A[j][i] = A[j][i] * 42;
```

Fetch $A[j+1][i]$, $A[j+2][i]$, $A[j+3][i]$, and $A[j+4][i]$



In Fortran, the elements of an array are stored in memory contiguously by column, and the original loop iterates over rows, potentially creating at each access a cache miss

A	B	C
D	E	F

is stored

A	D	B	E	C	F

Loop interchanging 2/2

Solution

This transformation switches the positions of one loop that is tightly nested within another loop.

```
int A[1024][1024];
for (int i = 1; i<1024; ++i)
  for (int j = 1; j<1024; ++j)
    A[j][i] = A[j][i] * 42;
```

Legal if the outermost loop does not carry any data dependence

Try it by yourself in gcc -floop-interchange

Tabular Grouping 1/2

Consider the following code, and direct mapping cache:

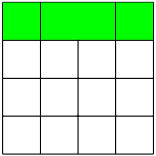
```
int A[1024]; int B[1024];
for (int j = 1; j<1024; ++j)
   A[j] = B[j] * 42;
```

Tabular Grouping 1/2

Consider the following code, and direct mapping cache:

```
int A[1024]; int B[1024];
for (int j = 1; j<1024; ++j)
   A[j] = B[j] * 42;
```

Fetch B[i], B[$i+1$], B[$i+2$] and B[$i+3$]

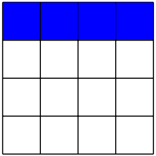


Tabular Grouping 1/2

Consider the following code, and direct mapping cache:

```
int A[1024]; int B[1024];
for (int j = 1; j<1024; ++j)
   A[j] = B[j] * 42;
```

Fetch A[i], A[$i+1$], A[$i+2$] and A[$i+3$]

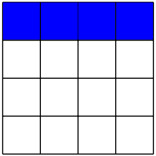


Tabular Grouping 1/2

Consider the following code, and direct mapping cache:

```
int A[1024]; int B[1024];
for (int j = 1; j<1024; ++j)
   A[j] = B[j] * 42;
```

Fetch A[i], A[$i+1$], A[$i+2$] and A[$i+3$]



Dynamic allocation does not allow padding. In the worst case, two miss per iterations

Tabular Grouping 2/2

Solution

Group the two tabular into one

```
struct twoval{int A; int B};
struct twoval R[1024];
for (int j = 1; j<1024; ++j)
   R[j].A = R[j].B * 42;
```

Avoid a lot of caches miss!

Very hard for compiler to detect such cases

Loop Blocking

Consider the code below.

```
int A[1024][1024]; int B[1024][1024];
for (int i = 1; i<1024; ++i)
  for (int j = 1; j<1024; ++j)
    A[i][j] = B[i][j];
```

- If A and B are aligned we may encounter problems.
- Similar problems occur when processing images: A[i][j] = B[i-1][j-1] + B[i-1][j] + B[i-1][j+1] + B[i][j-1] + B[i][j] + B[i][j+1] + B[i-1][j+1] + B[i+1][j] + B[i+1][j+1] ;
- In this latter case, padding is complicated...

Loop Blocking

Solution

Try to work with data that fit in memory!

```
int A[1024][1024]; int B[1024][1024];
for (int i = 1; i<1024; i += B)
   for (int j = 1; j<1024; j += B)
      for (int ii = 1; ii<min(1024, ii+B-1); ii += B)
         for (int jj = 1; jj< min(1024, ii+B-1); jj += B)
            A[i][j] = B[i][j];
```

Summary

- stalls in the processor can come from many reasons
  - from data dependencies between instructions
  - from instruction dependencies
  - from cache and memory
- modern compiler hardly try to reduce them
  - by using Instruction Level Parallelism (i.e, to have a lot of independent instructions)
  - all these optimization must occur before register allocation (which is the final step)
  - When writing a compiler, you must know the target processor by heart!
- Caches can be shared between many processors!