Optimisation de code : code reordering & loop unrolling

ARCHI 1

Daniela Genius
daniela.genius@lip6.fr

Les transparents seront accessibles sous
`https://www-soc.lip6.fr/~genius/`

Pipeline

Principle

- the treatment necessary for the execution of an instruction in cut into phases
- One pipeline stage par phase. 1 pipeline stage = 1 cycle and cycle = time taken by the longest stage
- At each cycle, one instruction starts its execution and does not wait for the end of execution of the previous instruction

Semantics

- Pipeline stages function in parallel
- Instructions enter sequentially, conform with the execution order imposed by the program
- In all phases, the order is preserved, instructions finish thus in the same order than the order they entered

Ideally, one instruction terminetes its execution at each cycle, but in practice, problem of dependencies.

Dependencies between instructions

Definition

Two instructions are dependent if one of them must be executed before the other, in order to keep the program correct.

Type of dependencies

- Data dependency : operands in common between 2 instructions.
- Control dependency : the first instruction is a branch, the execution of the following instruction depends on the result of the branch decision.

Data dependency

Definition

- RAW : Read After Write
  $i_1 \rightarrow_{RAW} i_2$
  i_1 writes Rx and i_2 reads Rx
  $\equiv$ use a result
- WAW : Write After Write
  $i_1 \rightarrow_{WAW} i_2$
  i_1 writes Rx and i_2 writes Rx
  $\equiv$ register reuse
- WAR : Write After Read :
  $i_1 \rightarrow_{WAR} i_2$
  i_1 reads Rx and i_2 writes Rx
  $\equiv$ register reuse

Examples

```
lw   $2, 0($4)
addi $5, $2, 10 # RAW
```

```
lw   $2, 0($4)
...
addi $2, $7, 10 # WAW
```

```
sw   $2, 0($4)
...
addi $2, $12, 10 # WAR
```

```
lw   $2, 0($4)
...
addi $4, $12, 10 # WAR
```

Inversing instructions of such couples in the code changes the program's semantics.

To succeed this course, you have to know the registers which are read and written by all instructions of the MIPS !

Control dependencies

- Execution of i_2 depends on the result of branch instruction i_1

```
bne R0, R6, loop
nop
add R2, R3, R4
```

- Delayed slot after each branch instruction : avoids having to cancel the instruction that entered into the pipeline before knowing the result of evaluating the branch condition
- Control dependencies do not cause stall cycles but limit performance (if delayed slot is filled with NOP)

Data dependencies and pipeline hazards

- Pipeline hazards occur when execution can change the order of read/write accesses of registers corresponding to operands
- The pipeline must be stalled to maintain correct execution
- MIPS32 :
  - operands are read from the register bank in the DECODE stage
  - Results are written to the register bank in the WB stage

Data dependencies and pipeline hazards

- WAW or Write After Write

```
lw   $2, 0($4)
addi $2, $7, 10 # WAW
```

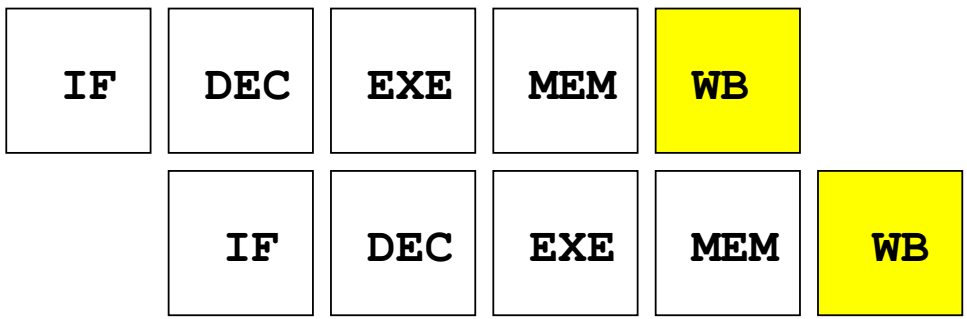
- Corresponds to writing 2 results to the same register

Data dependencies and pipeline hazards

- WAW or Write After Write

```
lw   $2, 0($4)
addi $2, $7, 10 # WAW
```

- Corresponds to writing 2 results to the same register

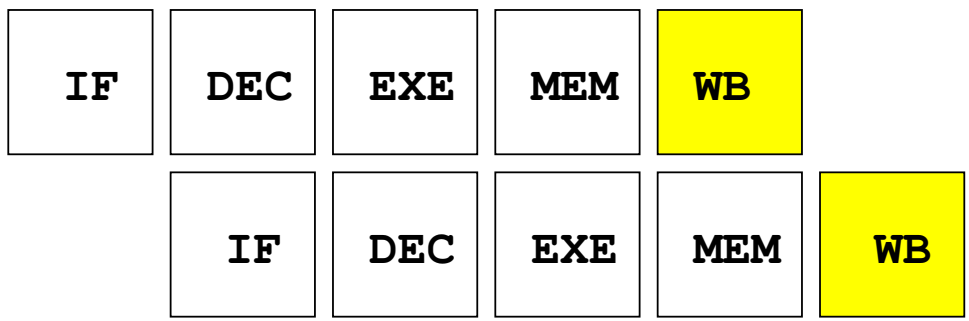


Data dependencies and pipeline hazards

- WAW or Write After Write

```
lw   $2, 0($4)
addi $2, $7, 10 # WAW
```

- Corresponds to writing 2 results to the same register



$\Rightarrow$ writes in order of the program

$\Rightarrow$ no problem in the pipeline

Data dependencies and pipeline hazards

- WAR or Write After Read

```
sw   $2, 0($4)
addi $2, $12, 10 # WAR
```

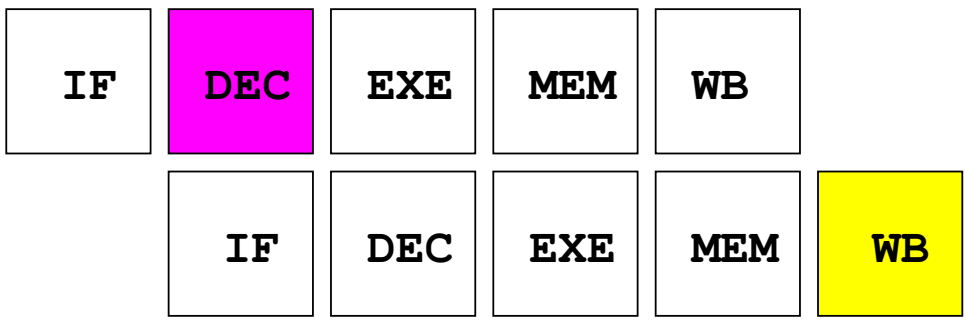
- Corresponds to writing to a register read by a precedent instruction :

Data dependencies and pipeline hazards

- WAR or Write After Read

```
sw   $2, 0($4)
addi $2, $12, 10 # WAR
```

- Corresponds to writing to a register read by a precedent instruction :

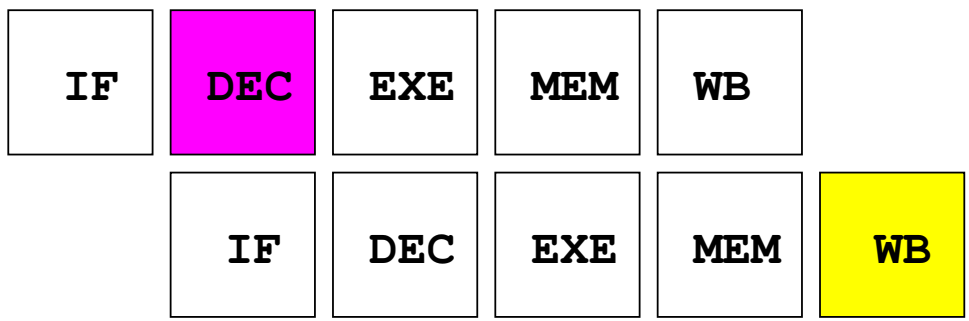


Data dependencies and pipeline hazards

- WAR or Write After Read

```
sw   $2, 0($4)
addi $2, $12, 10 # WAR
```

- Corresponds to writing to a register read by a precedent instruction :



$\Rightarrow$ the read from DEC occurs well before the dependent instruction writes its result to WB

$\Rightarrow$ no problem in the pipeline

Data dependencies and pipeline hazards

- RAW ou Read After Write

```
lw   $2, 0($4)
addi $5, $2, 10 # RAW
```

```
sub  $2, $4, $8
addi $5, $2, 10 # RAW
```

- Corresponds to using a result ; the dependent instruction reads the result of a preceding instruction. The latter has to be available, otherwise incorrect execution

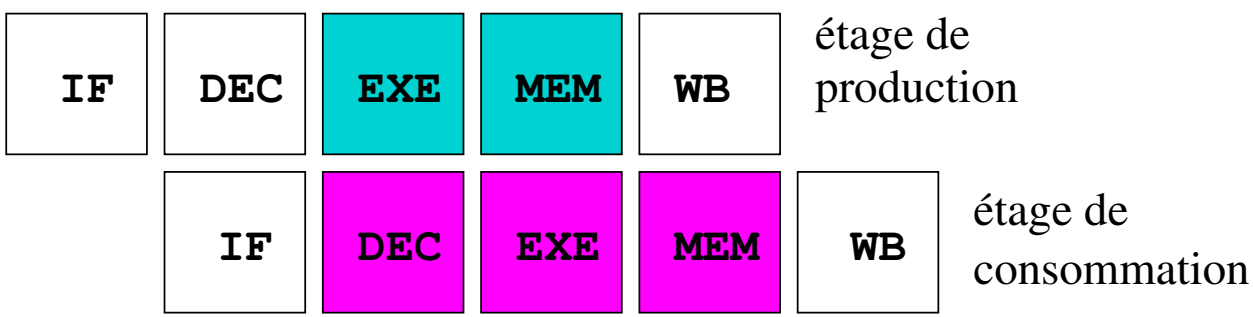
Data dependencies and pipeline hazards

- RAW ou Read After Write

```
lw   $2, 0($4)
addi $5, $2, 10 # RAW
```

```
sub  $2, $4, $8
addi $5, $2, 10 # RAW
```

- Corresponds to using a result ; the dependent instruction reads the result of a preceding instruction. The latter has to be available, otherwise incorrect execution



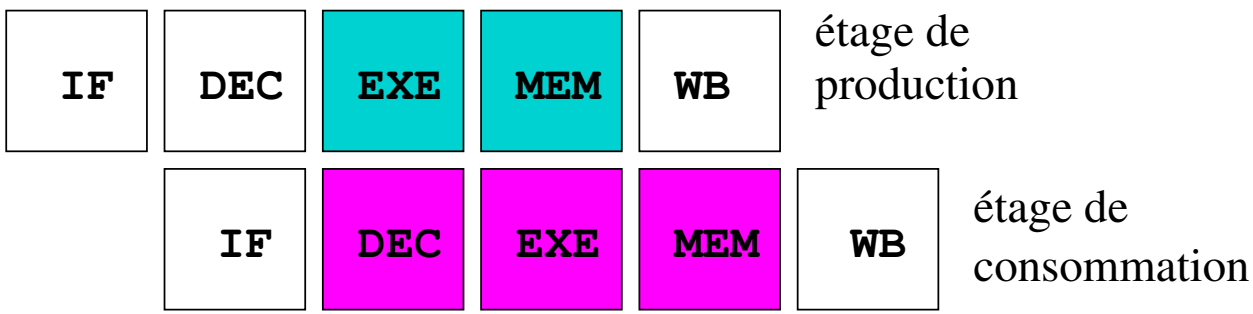
Data dependencies and pipeline hazards

- RAW ou Read After Write

```
lw   $2, 0($4)
addi $5, $2, 10 # RAW
```

```
sub  $2, $4, $8
addi $5, $2, 10 # RAW
```

- Corresponds to using a result ; the dependent instruction reads the result of a preceding instruction. The latter has to be available, otherwise incorrect execution



- If the operand is not availiable in the last stage (WB) where it can be taken (possibly different from the stage where it is consumed), the instruction is blocked at the entry of that stage and the entire pipeline is blocked upstream

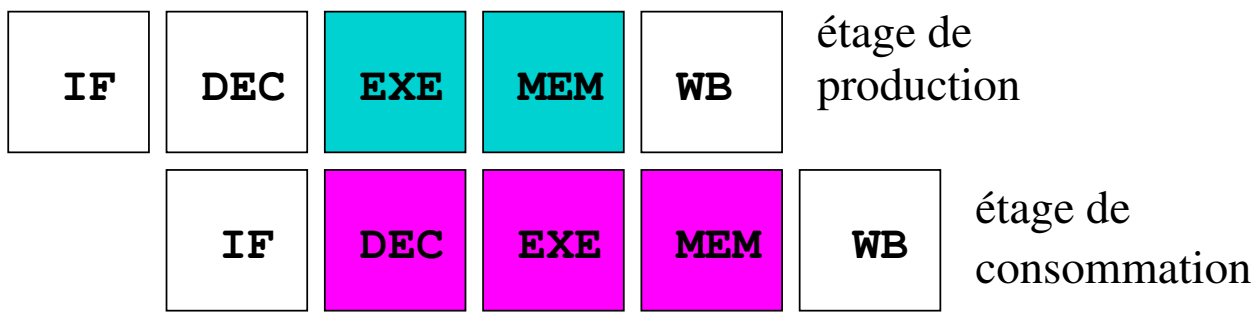
Data dependencies and pipeline hazards

- RAW ou Read After Write

```
lw   $2, 0($4)
addi $5, $2, 10 # RAW
```

```
sub  $2, $4, $8
addi $5, $2, 10 # RAW
```

- Corresponds to using a result ; the dependent instruction reads the result of a preceding instruction. The latter has to be available, otherwise incorrect execution



- If the operand is not availiable in the last stage (WB) where it can be taken (possibly different from the stage where it is consumed), the instruction is blocked at the entry of that stage and the entire pipeline is blocked upstream

$\Rightarrow$ Can introduce stall cycles into the pipeline

Data dependencies and pipeline hazards

- ALU operation : `opcod Rd, Rs, Rt`
  Production (`Rd`) at the end of EXE, consumed by (`Rs` and `Rt`) in the beginning of stage EXE

```
add $2, $4, $3          IF DEC EXE MEM WB
addi $5, $2, 10            IF  ??? ??? ???
```

- LOAD : `lw Rt, Imm(Rs)`
  Produces result (`Rt`) at the end of stage MEM, `addi` consumes its operands (`Rs` and `Imm`) at stage EXE

```
lw $2, 0($4)          IF DEC EXE MEM WB
addi $5, $2, 10          IF  ??? ??? ???
```

Data dependencies and pipeline hazards

- ALU operation : `opcod Rd, Rs, Rt`
  Production (`Rd`) at the end of EXE, consumed by (`Rs` and `Rt`) in the beginning of stage EXE

```
add $2, $4, $3          IF DEC EXE |MEM WB
addi $5, $2, 10            IF  DEC |EXE MEM WB
```

  There are 2 bypasses (for `Rs` and `Rt`) EXE $\rightarrow$ EXE : no stall cycles

- LOAD : `lw Rt, Imm(Rs)`
  Produces result (`Rt`) at the end of stage MEM, `addi` consumes its operands (`Rs` and `Imm`) at stage EXE
- `lw $2, 0($4)`

```
lw $2, 0($4)          IF DEC EXE MEM |WB
addi $5, $2, 10          IF  DEC  O  |EXE MEM WB
```

  There are 2 bypasses (for `Rs` and `Rt`) MEM $\rightarrow$ EXE : 1 stall cycle to obtain the operand

Data dependencies and pipeline hazards (2)

Branch : `beq Rs, Rt, label`
Consumption of `Rs` and `Rt` in the beginning of DEC

```
add $2, $4, $3        IF DEC  EXE  MEM  WB
beq $5, $2, loop         IF   ???  ???  ???
```

```
lw  $2, 0($3)       IF DEC  EXE  MEM  WB
beq $5, $2, loop       IF   ???  ???  ???
```

Data dependencies and pipeline (2)

Branch : `beq Rs, Rt, label`
Consumption of `Rs` and `Rt` in beginning of DEC

```
add $2, $4, $3        IF   DEC   EXE |MEM   WB
beq $5, $2, loop           IF     0   |DEC   EXE   MEM   WB
```

There are 2 bypasses (for `Rs` and `Rt`) EXE $\rightarrow$ DEC : 1 stall cycle to obtain the operand

```
lw  $2, 0($3)        IF DEC EXE MEM |WB
beq $5, $2, loop        IF   0   0  |DEC EXE MEM WB
```

There are 2 bypasses (for `Rs` and `Rt`)MEM $\rightarrow$ DEC : 2 stall cycles to obtain the operand

Data dependencies and pipeline hazards (3)

STORE : `sw Rt, Imm (Rs)`
Consumption of `Rs` in EXE, `Rt` in MEM

```
addi $2, $3, 1          IF DEC EXE MEM WB
sw   $2, 0($4)             IF  ??? ??? ???
```

```
lw   $2, 0($3)          IF DEC EXE MEM WB
sw   $2, 0($4)             IF  ??? ??? ???
```

Data dependencies and pipeline hazards (3)

- A bypass required multiplexers and corresponding control logic
- Critical stages : (MEM, IF, DEC, EXE)
- MEM is the most critical stage
- Cycle duration would be prolonged if put into MEM, used only for STORE
- No bypass in the MEM stage
- A STORE must obtain the value of the operand to be written to memory in DEC or EXE
- EXE is the first stage where it is possible to obtain the operand to be written to memory, even of operand consumed in MEM !

Data dependencies and pipeline hazards (3)

- A bypass required multiplexers and corresponding control logic
- Critical stages : (MEM, IF, DEC, EXE)
- MEM is the most critical stage
- Cycle duration would be prolonged if put into MEM, used only for STORE
- No bypass in the MEM stage
- A STORE must obtain the value of the operand to be written to memory in DEC or EXE
- EXE is the first stage where it is possible to obtain the operand to be written to memory, even of operand consumed in MEM !

Data dependencies and pipeline hazards (3)

STORE : `sw Rt, Imm (Rs)`
Consumption of `Rs` in EXE, `Rt` in MEM

```
addi $2, $3, 1          IF  DEC  EXE |MEM  WB
sw  $2, 0($4)               IF   DEC |EXE  MEM WC
```

Bypass EXE $\rightarrow$ EXE (for `Rt` of `sw`)

```
lw  $2, 0($3)         IF DEC EXE MEM |WB
sw  $2, 0($4)            IF  DEC  0  |EXE MEM WB
```

Bypass MEM $\rightarrow$ EXE (for`Rt` du `sw`) : 1 stall cycle

Watch out for STORE !!!

Test

```
lw   $4, 0($3)          IF   DEC   EXE   MEM   WB
lw   $2, 0($4)               IF    ???   ???   ???
```

Test

```
lw   $4, 0($3)          IF   DEC   EXE MEM |WB
lw   $2, 0($4)               IF    DEC   0   |EXE   MEM WB
```

Adresses calculated in EXE...

Bypass MEM $\rightarrow$ EXE (for `Rs` of 2nd `lw`)

Code example

C Code :

```
int a[size];
...
for (i = 0; i != size; i++)
  a[i] = 2 * a[i];

...
```

Compiled assembler code :

```
 # i in R8
 # size in R6, a[] in R5

   xor  R8, R8, R8
   beq  R6, R0, suite
   sll  R9, R6, 2   # size * 4
   add  R9, R9, R5  # @a[size]

loop:
   lw   R4, 0(R5)   # R4 <- *tab
   sll  R7, R4, 1   # R7 * 2
   sw   R7, 0(R5)   # tab[i] <- R7
   addiu R5, R5, 4 # tab++
   bne  R9, R5, loop
```

Assembler code executable on MIPS32 ?

Code example

Compiled Assembler code :

```
# i in R8
# size in R6, a[] in R5

  xor  R8, R8, R8
  beq  R6, R0, suite
  sll  R9, R6, 2  # size * 4
  add  R9, R9, R5 # @a[size]

loop:
  lw   R4, 0(R5)
  sll  R7, R4, 1
  sw   R7, 0(R5)
  addiu R5, R5, 4
  bne  R9, R5, loop
```

MIPS32 assembler code :

```
   # i in R8
   # size in R6, a[] in R5
    xor  R8, R8, R8
    beq  R6, R0, suite
    NOP      # delayed slot
    sll  R9, R6, 2 # size * 4
    add  R9, R9, R5 # @a[size]

loop:
    lw   R4, 0(R5)
    sll  R7, R4, 1
    sw   R7, 0(R5)
    addiu R5, R5, 4
    bne  R9, R5, loop
    NOP      # delayed slot
```

Code analysis (blackboard)

- Simplified scheme to determine the number of cycles per iteration
- Calculate the CPI
- Calculate the useful CPI

- How many stall cycles ? Can we do better ?

Instruction reordering

- Cover les delays between dependent instructions by the execution of independent instructions
- One can change the order of instructions AS LONG AS ALL dependencies (RAW, WAR, WAW) are respected
- One can also change the order of instructions to avoid ressource conflicts (delays due to conflicts not to dependencies cf. MIPS SS2)

- Example on blackboard

Reordering

- Allows reducing execution time and limiting stall cycles
- Importance also to limit empty delayed slots (avoid NOPs)

```
# R5 = @tab[i], R9 = @tab[
    size]
loop:
 lw   R4, 0(R5)
 sll  R7, R4, 1
 sw   R7, 0(R5)
 addiu R5, R5, 4
 bne  R9, R5, loop
 nop
```

- 1 stall cycle between `lw` and `sll`
- 1 stall cycle between `addiu` and `bne`
- 1 useless `nop`
- 6 instructions + 2 stall cycles = 8 cycles/iteration for 5 useful instructions

Reordering

- Reordering to limit stall cycles + `nop`
- WAR dependency between `addiu` and `sw` but correction possible as increment of R5 has constant step (+4) de R5 and immediate in `addiu` $\rightarrow$ raise the `addiu` before the `sw` (and adjust immediat of `sw`)
- Put `sw` in the delayed slot

```
# R5 = @tab[i], R9 = @tab[
    size]
loop:
 lw   R4, 0(R5)
 sll  R7, R4, 1
 sw   R7, 0(R5)
 addiu R5, R5, 4
 bne  R9, R5, loop
 nop

# 6 instructions
# 2 stall cycles, 1 nop
```

```
# R5 = @tab[i], R9 = @tab[N
    ]
loop:
 lw   R4, 0(R5)
 addiu R5, R5, 4
 sll  R7, R4, 1
 bne  R9, R5, loop
 sw   R7, -4(R5)
#5 instructions : 0 stall
    cycle, 0 nop
#5 instructions : 2 for
    loop management, 3 for
    the body !
```

Loop unrolling

- The additional cost of loops can be reduced by unrolling : several iteration share the cost
- The loop body becomes bigger : more opportunity for reordering
- Unrolling by factor `u` = the loop unrolled `u` times performs `u` iterations of the original loop
- High level illustration :

```
   /* original loop */
for(i = 0 ; i<N ; i++)
  tab[i] = tab[i]*2;
```

```
   /* unrolled loop */
for(i = 0 ; i+1<N ; i+=2){
   tab[i] = tab[i]*2;
   tab[i+1] = tab[i+1]*2;
}

   /* remainder */
for( ; i< N; i++)
   tab[i] = tab[i]*2;
```

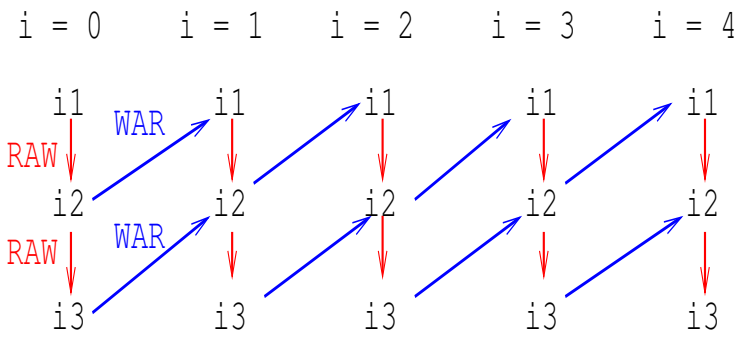
Loop unrolling

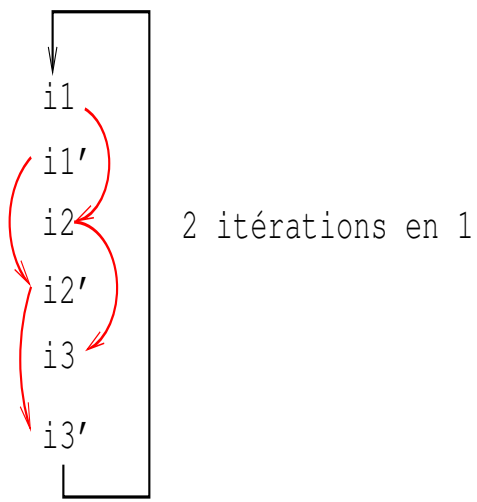
Example on blackboard.

Loop unrolling : a different view

```
Loop:
     lw R4, 0(R5)        #i1  |
     sll R7, R4, 1       #i2  | corps de la boucle
     sw R7, 0(R5)        #i3  |
     addiu R5, R5, 4     #i4  |
     bne R5, R9, Loop    #i5  | gestion de boucle
     nop                 #i6  |
```

Déroulage : renommage de registres pour
éliminer les dépendances WAR entre 2 (ou plus) itérations
pour entrelacer les instructions des itérations





Optimisation and control flow

```
# R5 = @tab[i], R7 = @tab[N]
loop:
    lw   R8, 0(R5)   # read tab[i]
    bgez R8, endif   # if tab[i] positive or zero continue at
        enfif
    nop
    sub  R9, R0, R8 # R9 = -R8
    sw   R9, 0(R5)   # tab[i] = -tab[i]
endif:
    addiu R5, R5, 4  # @tab[i]++ address of next element
    bne  R7, R5, loop # loop if @tab[i] != @tab[N]
    nop
```

- branch in loop... translates an if...then
- problem for reordering ?
- problem for loop unrollling ?
- need to analyze the control flow...

Basic block : definition and calculation

Definition

- Basic block : sequence of instructions with a single entry point (1st instruction) and single exit point (last instruction) : if the first instruction is executed, so are all

Determining BB

1. Determine the headers
   - first instruction of a function/a code scrap
   - instruction following the last delayed slot after a jump
   - instruction which is target of a jump
2. Linear determination of BB : from one header to the next (non inclusive).

Partitioning into basis blocks : example

```
# R5 = @tab[i], R7 = @tab[N]

loop:------- BB1 ------
    lw   R8, 0(R5)
    bgez R8, endif
    nop
    ------- BB2 ------
    sub  R9, R0, R8
    sw   R9, 0(R5)
    ------- BB3 ------
endif:
    addiu R5, R5, 4
    bne  R7, R5, loop
    nop
```

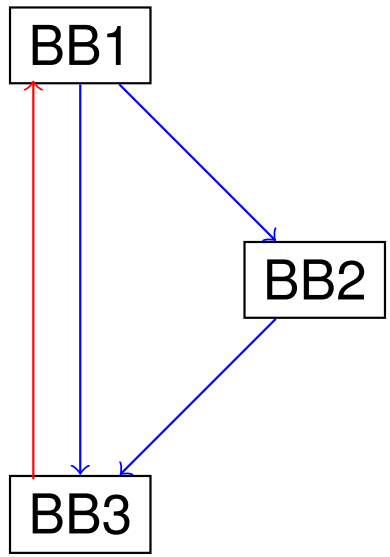
CFG : Control Flow Graph

- The links between the BB translate the control flow of the program
- CFG : graph reflecting the control flow
- There is an arc between two blocks B1 and B2 if
  - there is a jump from B1 to B2
  - if B2 follows B1 in the order of the programme and B1 doeas not terminate by an inconditional jump (j label)

CFG : example

```
# R5 = @tab[i], R7 = @tab[N]

loop:------- BB1 ------
    lw   R8, 0(R5)
    bgez R8, endif
    nop
    ------- BB2 ------
    sub  R9, R0, R8
    sw   R9, 0(R5)
    ------- BB3 ------
endif:
    addiu R5, R5, 4
    bne  R7, R5, loop
    nop
```



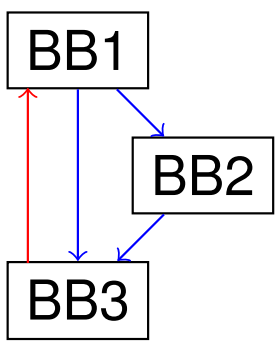
Optimizations and control flow

- Reordering : pay attention to control flow !
- If single basic block : by definition of BB, the instruction order can be changed (respecting dependenciesa) without regard to the flow
- If several basic blocks : more complex because the control flow has to be taken into account

Reordering and control flow : example

```
# R5 = @tab[i], R7 = @tab[N]

loop:------- BB1 ------
    lw   R8, 0(R5)
    bgez R8, endif
    nop
    ------- BB2 ------
    sub  R9, R0, R8
    sw   R9, 0(R5)
    ------- BB3 ------
endif:
    addiu R5, R5, 4
    bne  R7, R5, loop
    nop
```



- BB1 and BB3 always executed : correct to lower instructions of BB1 into BB3, or lift instructions of BB3 into BB1 (if all dependencies are respected)
- BB2 nor always executed : incorrect to lower an instruction of BB1 into BB2, neither lift an instruction of BB3 to BB2 if no possible execution !
- Correct : lift instructions from BB2 into BB1 ou lower instructions from BB2 into BB3 if semantics unchanged...

Loop unrolling with internal control

```
#R5 = @tab[i], R7 = @tab[N]

loop:------- BB1 ------
    lw   R8, 0(R5)
    bgez R8, endif
    nop
    ------- BB2 ------
    sub  R9, R0, R8
    sw   R9, 0(R5)
    ------- BB3 ------
endif:
    addiu R5, R5, 4
    bne  R7, R5, loop
    nop
```

- How to unroll the loop ?
- Two execution paths possible per iteration
- Loop unrolling must preserve the different possible combinations of paths in the unrolled loop body !
- Copy labels, serialize part of the treatment...
- Example on blackboard

Loop unrolling and (in)dependency of iterations

```
for (i = 0 ; i < N ; i++)
  tab[i] = tab2[i] + tab[i];
```

```
for (i = 0; i+1 < N ; i += 2) {
  tab[i] = tab2[i] + tab[i];
  tab[i+1] = tab2[i+1] + tab[i+1];
}
```

```
for (i = 1; i < N ; i++)
  tab[i] = tab[i-1]+ 2*tab[i];
```

```
for (i = 1 ; i+2 < N ; i += 3) {
   tab[i] = tab[i-1] + 2*tab[i];
   tab[i+1] = tab[i] + 2*tab[i+1];
   tab[i+2] = tab[i+1] + 2*tab[i+2];
}
```

- Independence of iterations versus dependance
- Duplication interlacing versus reutilization of calculations / correction of unrolled version
- Loop unrolling must ensure the use of correct values : in the 2nd assignment of the unrolled version `tab[i]` modified, `tab[i+1]` not modified !
- No parallelization of treatments but "pipeline" and reutilization of available values : new value of which one cannot do the 2nd assignment before the calculation of `tab[i]` but `tab[i]` need nor be re-read from memory

Conclusion

- Code can be optimized to improve execution
- Reordering : change instruction order while respecting data dependencies and control flow !
- Loop unrolling : n replications of the loop body ans optimisation of the unrolled body, attentive to inter-iteration dependencies + control flow in each iteration.
- Second lesson : software pipelining