Study Guide ER1 ARCHI

Written by : Jorge Mendieta

Battle Plan

Below is a Battle Plan designed to tackle the first ER1 of Architecture des processeurs RISC (ARCHI)

Exams are usually structured in 3 parts:

1. Non-Superpipelined & Superpipelined processors
2. Assembly code
3. Code optimization

First part: Non-Superpipelined & Superpipelined processors

Recommendations

- It is **highly recommended** to do this method at the beginning of the exam.
- Remember that the **critical path** is our slowest stage in the pipeline, usually instructions that acces the memory (IFC and MEM)
- **Note: Read closely** and pay attention to the specifics of the processor of the exam.
    - *ER1 2022 example*:
        - "No bypass at beginning of MEM1 stage."
        - "In addition, unlike Mips, there is no bypass to retrieve the result of a memory access instruction when an instruction depends on a *load instruction*."

Pipeline rules

1. Data can only pass from one stage to another using a register.
2. Stages need to be time-balanced.
3. Every component belongs to one and only one stage (only one stage can write to the register bank)

Simplified Diagram (MIPS 32 Pipelined to 5 stages example)

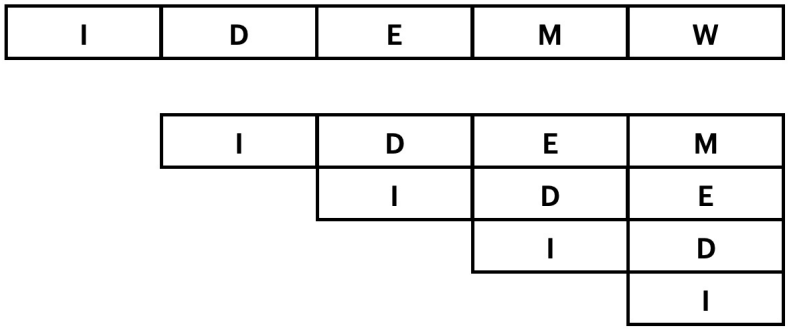
1. Draw each stage of the simplified diagram (e.g. I, D, E, M, W).
    - Pay attention to the **superpipeline specifications** (e.g. IF1, IF2, DEC, EX1, EX2, MM1, MM2, WBK)

    Mips 32

2. Continue drawing the subsequent stages of the pipeline until when the last instruction's IFC stage begins at the first instruction's WBK (further stages are not needed to calculate what's next), creating sort of an "inverted staircase".

    Mips 32

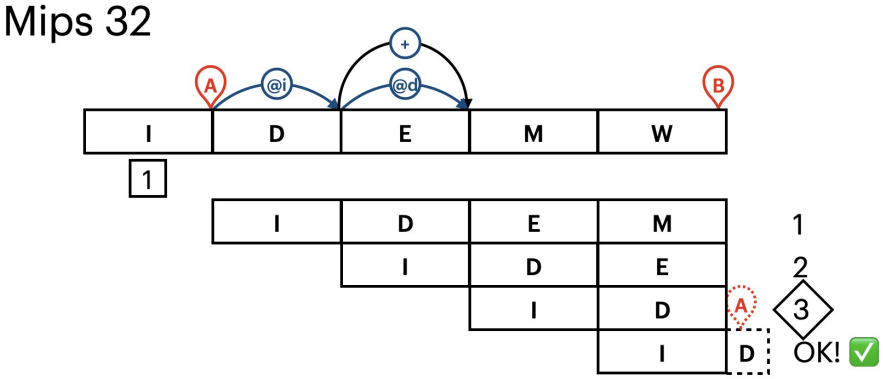
    

3. Mark the 5 strategic points (**A, B, @i, @d, +**)
    1. **Point A** : Récupération des opérandes depuis la banc de registre (**au plus tôt**)
    2. **Point B** : Dernièr étage pouvant écrire dans la banc de registre (**au plus tard**)
        - Once **Point A** passes **Point B**, we're "safe" and we won't have any more data dependencies since the register bank has already been written.
        - The **number** of instructions until **Point A** passes **Point B after the first instruction** marks the **maximum order of data dependencies** (Denoted by ◇)
    3. **@i =** Calcul d'adresse de la prochaine instruction (Branch Instructions)
        - Usually DEC **unless specified**
        - The previous stage before **@i** is finished is our **number of delayed slots** (denoted by □)
    4. **@d =** Calcul d'adresse pour les accèss mémoire pour les données

- Usually EXE

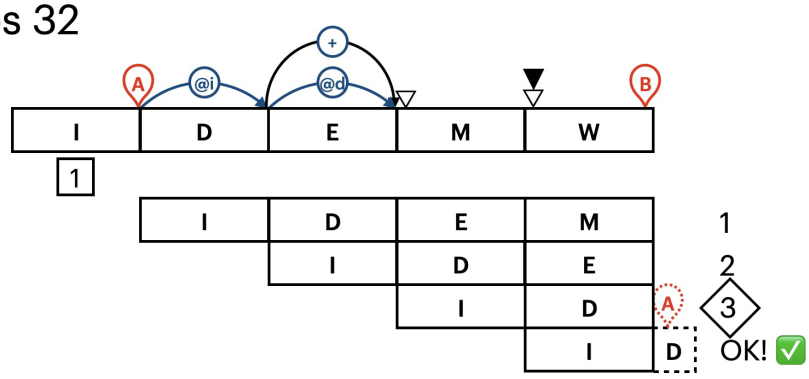
5. **+** = ALU execution
   - Usually EXE

Mips 32

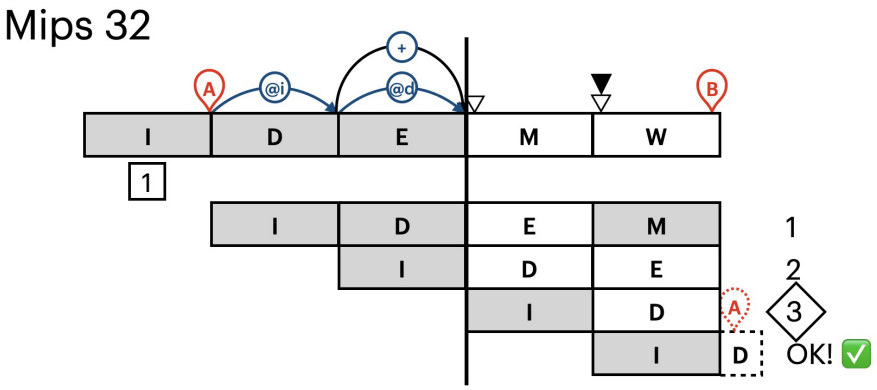


4. Search the **Producers** (Denoted by ▽,▼)
   - Produce a result to the register bank
   1. Note down the stage where the result is available **the earliest** until **point B**
      1. **Arithmetic & Logic** Instructions - Denoted by ▽
         1. End of EXE (End of the ALU execution stage marked by **point (+)**)
      2. **Load** instructions - Denoted by ▼
         1. End of MEM
5. Mark the **producer** symbols from the **earliest available stage** → **point B** (e.g. ▽, …, ▽, B)

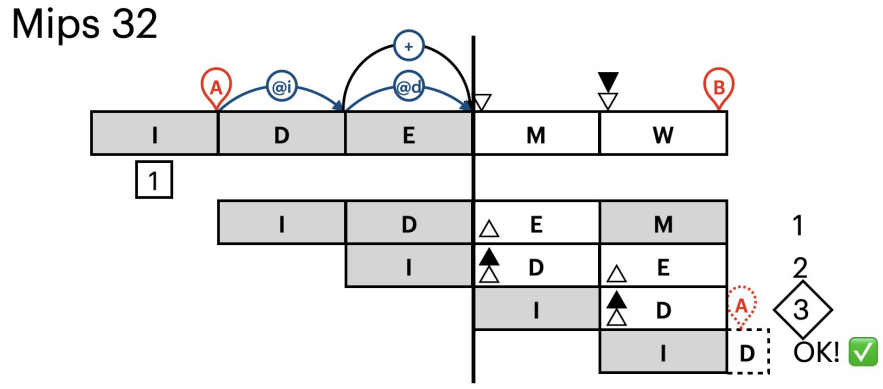
Mips 32



6. Start **reducing the available options** for bypasses to find our working space:
   1. Mark a line on the first producer (▽). Before this line there can't be any bypasses since the data won't be yet available, so we eliminate them from the bypass options.
   2. IFC instructions can't have bypasses, eliminate them
   3. Usually, there can't be any bypasses on **entry** of MEM stages unless explicitly stated otherwise (**critical path**), so eliminate them



7. Search the **consumers** (Denoted by △, ▲, or different variations)
    - Consume a result from the register bank
    1. Note down the stage from **point A** until where it's available **the latest** before the instructions begins and consumes the needed operand
    2. **Note:** Instructions sharing the same configuration (same consuming point) will share the same symbol
        1. **Arithmetic & Logic Instructions** - Denoted by △
            1. Beginning of EXE (Beginning of **point +**)
        2. **Store/Load instructions** - Denoted by △
            1. Beginning of EXE
            2. *Beginning of E1 (**@d** - Superpipelined with EXE1-EXE2)*
            3. *Beginning of E2 (Data - Superpipelined with EXE1-EXE2)*
        3. **Load instructions** - Denoted by △
            1. Beginning of EXE
        4. **Branch instructions** - Denoted by ▲
            1. Beginning of the stage where the **@i** is computed (Computing of the address of the next instruction)
8. Mark down the **consumer** symbols from **the latest available stage** <- **Point A**(e.g. A, △, …, △)



9. Note down the **number of operands** (Rs, Rt) consumer instructions affect. (e.g. △2, ▲1)

    1. **Note:** Pay attention if there are any forbidden bypasses

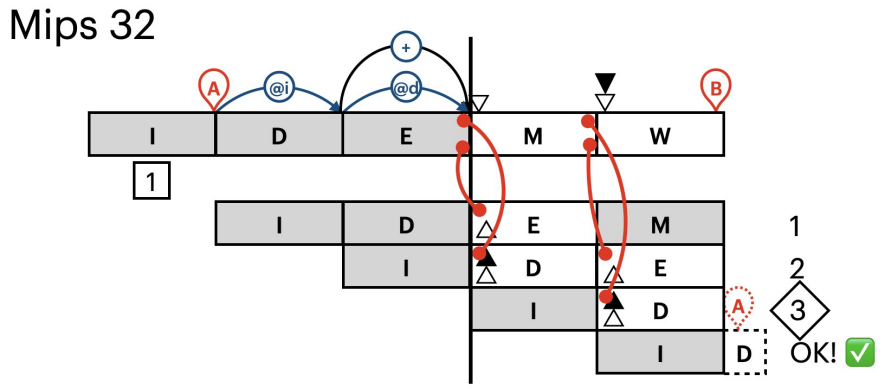
Notes

- Splitting the Decode stage can lead to additional delayed slots.
- The compiler has about 80% chance to replace a delayed slot for an useful instruction.
- Load instruction bypasses for produced data are rarely used (*Ahmdal's law*), ergo, sometimes they're not implemented.

Bypasses

From this point we can Mark the bypasses by connecting the **producers** to the **consumers**

1. Bypasses for Branches (▲) and Loads (▼) are **obligatory**
2. Given the same configuration (▽ to △) and order for a bypass for different stages, we can choose what fits best our needs
    1. **Note:** It is recommended not to add multiple bypasses to the same stage if possible. This is to prevent adding additional HW (e.g. multiplexers) and affecting the signal speed through the stage. This is even more important for the **critical path**.
3. The **total number of bypasses** is found by counting the different bypasses for each operand (e.g. ▽ → △2 means there are two bypasses (one for each operand Rs & Rt))
    1. In the case of a classic MIPS 32 5-stage pipeline, there are **8 bypasses in total**



Give examples of every type bypass there is:
2. Circle the operands (Rs, Rt) to mark the dependencies
3. Specify which operands the bypass affects (Rs, Rt, or both)
4. Specify the stages of the bypass (e.g. EXE → MEM)

Note: Don't forget the wavy lines "~~" to denote arbitrary instructions for higher-order dependencies

Example from 2022's ER1

```
# Order 3 dependency: (EXE2 -> DEC)
Add (R10), R0, R4
# ~~~~
# ~~~~
Add R10, (R10), (R10) # Rs & Rt
```

Important Note: When we have the choice to add a bypass between various options, it's better to not affect any more a stage that has already a bypass added. This is due to the **critical path** because there is additional HW the signal has to pass through, making the path slower.

Ergo, it is very important that we create a solution with a **balanced pipeline**, since the performance is tied to the slowest stage of the pipeline.

Notes

- IFC stage does not consume anything from the registry bank, hence there aren't any bypasses for this stage.
- In general the MEM stage should not have any bypasses since it is usually the **critical path**, which we don't want to add additional HW making the signal even slower.

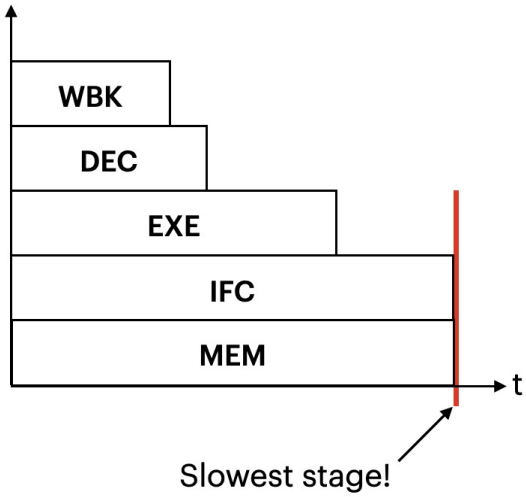
Hardware & Software Solutions

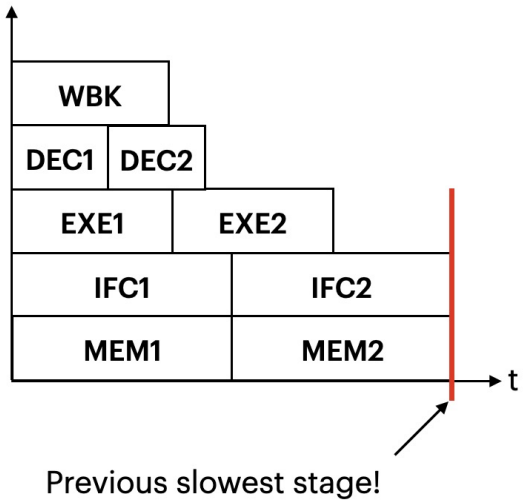
Hardware Solutions	Software Solutions
Bypasses	Compiler (Delayed slots `NOP`)
Stall cycles (Cycles de gel)	Optimization (e.g. loop unrolling, reordering)

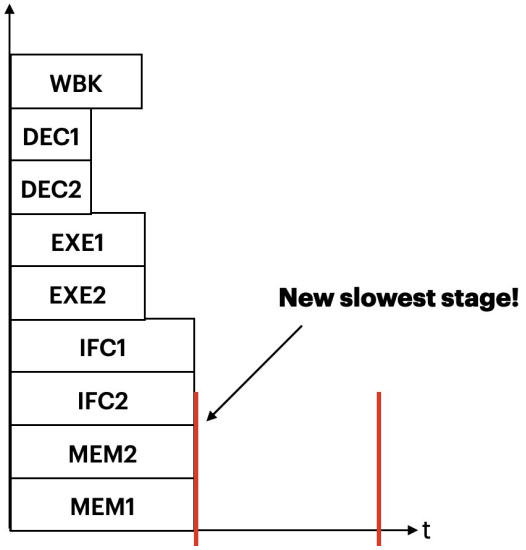
- The compiler makes all the optimizations, the processor only executes the instructions
- For example, if we take away the stall cycles, we need to remplace that for a SW solution. In that case we would need to add additional `NOP` instructions.

Why split into multiple stages?

To increase frequency and reduce the critical path!







Second part: Assembly code

Adapting code for MIPS execution

1. As usual, we need to identify the dependencies for **consumers** and note which operands it depends on (Rs, Rt, or both)
2. Count the stall cycles for each dependencies (if any)
3. Add any delayed slots (`NOP` instructions) after Branch instructions

Afterwards, we can obtain three important metrics:

1. **# Instructions :** Simply count the total number of ASM instructions (including `NOP`)
2. **# Useful Instructions :** Total number of instructions without `NOP`
3. **# Cycles :** Obtained by adding the *# Instructions* and the *Stall cycles*

With these three pieces of data, we can do more calculations to obtain three metrics:

1. Cycles per instruction - CPI
2. Cycles per useful instruction - CPI_{utile}
3. Performance - $Performance$

$$CPI = \frac{\#\,Cycles}{\#\,Instructions}$$

$$CPI_{useful} = \frac{\#\,Cycles}{\#\,Instructions_{useful}}$$

$$Performance = \frac{Freq_{processor}}{CPI}$$

Comparing performance between processors

Measuring Frequency needed to surpass Processor 1's performance with Processor 2:

$$Performance_1 = \frac{Freq_1}{CPI_1}$$

$$Performance_2 = \frac{Freq_2}{CPI_2}$$

$$Performance_2 > Performance_{21}$$

$$\rightarrow \frac{Freq_2}{CPI_2} > \frac{Freq_1}{CPI_1}$$

$$\rightarrow Freq_2 > \frac{CPI_2}{CPI_1} \cdot Freq_1$$

$$\rightarrow Freq_2 > \frac{\frac{Cycles_2}{Instructions_2}}{\frac{Cycles_1}{Instructions_1}} \cdot Freq_1$$

Finally, we obtain the final frequency

$$\rightarrow Freq_2 > \frac{Cycles_2 \cdot Instructions_1}{Instructions_2 \cdot Cycles_1} \cdot Freq_1$$

Third part: Code optimization

Dependencies

1. Draw **dependency graph** with various columns
    1. Each column is a series of instructions that depend on one from another
    2. Instructions can have both child dependencies and parent dependencies or only child dependencies
    3. Instructions with parent dependencies cannot be moved higher than their parent, or else it would break the context of the code and corrupt it
2. "Break" dependencies by reorganizing the order of the instructions
    1. This won't really eliminate the dependency but instead make use of the available bypasses and eliminate the stall cycle
3. Replace delayed slots (`NOP`) after Branches for instructions without parent dependencies

Loop unrolling

Note :

- Only `for` loops can be unrolled

Loops can be divided in two parts or *blocks*

1. **Treatment** (Traitement)
    - Executes the algorithm per se
2. **Management** (Gestion)
    - Updates the pointers for the next iteration

The **Treatment block** can be duplicated to execute the algorithm twice per iteration

- Copy the whole **treatment block after** the first treatment block instead of intercaling instructions line by line
- For the second **treatment block**, we can use the register + 10 (ex. R9 -> R19, R12 -> R22)
    - Pay attention to the special registers! (Ex. R31)
- After this, we can start reordering instructions by inserting items 1 by 1 like a zipper to break dependencies

For the **Management block** we need to update the pointers and steps accordingly

- If the treatment block executes twice per iteration, then the counter needs to advance twice as fast
- If an instruction in the management block is moved outside to break a dependency, then take care to update the pointer accordingly (original value - byte value(b, h, w)) to avoid modifying the wrong data

Note :

- Pay attention to the Load/Store instructions to mantain correct **memory alignment**
  - `LB` or `SB`: Multiple of single Bytes (1 by 1) - **Self aligned!**
  - `LH` or `SH`: Multiple of two Bytes (2 by 2)
  - `LW` or `SW`: Multiple of four Bytes (4 by 4)

Note :

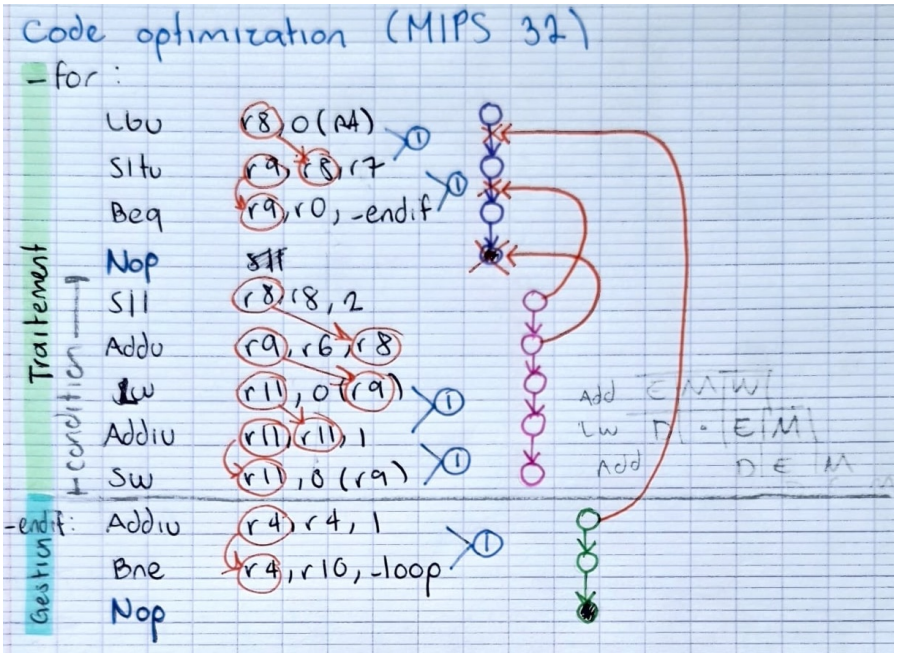
- If a register remains unchanged in the **treatment block** it can be a constant value. We can keep that same register when copying the treatment block for loop unrolling.

Example ER1 2017

Code before optimization

```
# Version 1
_loop:
    # Traitement
    lbu   r8,  0(r4)       # lecture img[i]
    sltu  r9,  r8, r7      # img[i] < seuil ?
    beq   r9,  r0, _endif
    nop                    # Delay slot - Adapted for MIPS32
    sll   r8,  r8, 2
    addu  r9,  r6, r8      # calcul &histo[img[i]]
    lw    r11, 0(r9)       # lecture histo[img[i]]
    addiu r11, r11, 1
    sw    r11, 0(r9)       # écriture histo[img[i]]
_endif:
    # Gestion
    addiu r4,  r4, 1
    bne   r4,  r10, _loop
    nop                    # Delay slot - Adapted for MIPS32
```

Dependency graph



Code after reodering

```
# Version 1
_loop:
    # Traitement
    lbu   r8,  0(r4)
    addiu r4,  r4, 1       # Reordered
    sltu  r9,  r8, r7
    sll   r8,  r8, 2       # Reordered
    beq   r9,  r0, _endif
    addu  r9,  r6, r8      # Reordered - Replaced delay slot (nop)
    lw    r11, 0(r9)
    addiu r11, r11, 1
    sw    r11, 0(r9)
_endif:
    # Gestion
    bne   r4,  r10, _loop
    nop
```

Unrolled loop (without previous reordering)

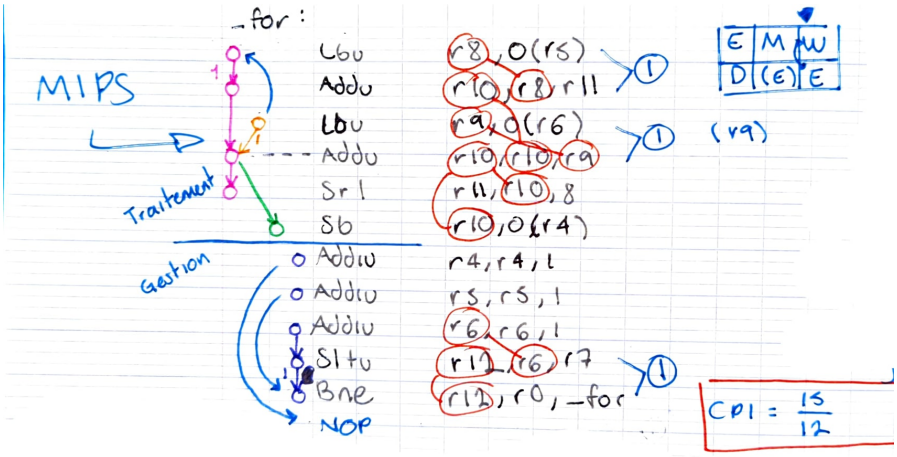
```
# Version 1
_loop:
    # Traitement 1
    lbu   r8,  0(r4)
    # Traitement 2
    lbu   r18, 1(r4)    # Advance index by 1
    # Traitement 1
    sltu  r9,  r8, r7
    # Traitement 2
    sltu  r19, r18, r7
    # Traitement 1
    beq   r9,  r0, _endif
    nop
    sll   r8,  r8, 2
    addu  r9,  r6, r8
    lw    r11, 0(r9)
    addiu r11, r11, 1
    sw    r11, 0(r9)
_endif1:
    # Traitement 2
    beq   r19, r0, _endif2
    nop
    sll   r18, r18, 2
    addu  r19, r6, r18
    lw    r21, 0(r19)
    addiu r21, r21, 1
    sw    r21, 0(r9)
_endif2:
    # Gestion
    addiu r4, r4, 2       # Add 2 instead of 1
    bne   r4, r10, _loop
    nop
```

Example ER1 2022

Code before optimization

```
_for:
    # Traitement
    lbu   r8,  0(r5)
    addu  r10, r8, r11
    lbu   r9,  0(r6)
    addu  r10, r10, r9
    srl   r11, r10, 8
    sb    r10, 0(r4)
    # Gestion
    addiu r4,  r4, 1
    addiu r5,  r5, 1
    addiu r6,  r6, 1
    sltu  r12, r6, r7
    bne   r12, r0, _for
    nop
```

Dependency graph



Code after reodering

```
_for:
    # Traitement
    lbu   r8,  0(r5)
    lbu   r9,  0(r6)    # Reordered
    addu  r10, r8, r11
    addu  r10, r10, r9
    srl   r11, r10, 8
    sb    r10, 0(r4)
    # Gestion
    addiu r6,  r6, 1
    sltu  r12, r6, r7
    addiu r5,  r5, 1    # Reordered
    bne   r12, r0, _for
    addiu r4,  r4, 1    # Reordered - Replaced delay slot (nop)
```

Unrolled loop (without previous reordering)

```
_for:
    # Traitement 1
    lbu   r8,  0(r5)
    addu  r10, r8, r11
    lbu   r9,  0(r6)
    addu  r10, r10, r9
    srl   r11, r10, 8
    sb    r10, 0(r4)
    # Traitement 1
    lbu   r18, 1(r5)       # Advance index by 1
    addu  r20, r18, r21
    lbu   r19, 1(r6)       # Advance index by 1
    addu  r20, r20, r19
    srl   r21, r20, 8
    sb    r20, 1(r4)       # Advance index by 1
    # Gestion
    addiu r4,  r4, 2       # Add 2 instead of 1
    addiu r5,  r5, 2       # Add 2 instead of 1
    addiu r6,  r6, 2       # Add 2 instead of 1
    sltu  r12, r6, r7
    bne   r12, r0, _for
    nop
```