Títol: Dynamic vectorization of instructions

Volum: 1

Alumne: Aaron Call Barreiro

Director/Ponent: Xavier Martorell Bofill

Departament: Arquitectura de computadors

Data: *30*/05/2014

DADES DEL PROJECTE

Títol del Projecte: Dynamic vectorization of instructions

Nom de l'estudiant: Aaron Call Barreiro
Titulació: Enginyeria en Informàtica
Crèdits: 37.5
Director/Ponent: Xavier Martorell Bofill
Departament: Arquitectura de Computadors

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President: Ramon Canal Corretger

Vocal: Isabel Navazo Alvaro

Secretari: Xavier Martorell Bofill

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data:

Table of contents

List of figures

Code fragments

1. Introduction and motivation

1.1 Context

Nowadays computers consume large amounts of energy, mainly due to microprocessors, and this have became a serious problem, not only because of pollution but also because of money and mobile devices.
On one hand, datacenters and supercomputers are large sets of computers, the smallest one can be of 2.000 computers side by side, this means a lot of energy which due to conservation of energy law, it transforms to a lot of hot. So much hot in a reduced space may damage wires and other devices that are essential part of these buildings, so these hot have to be dissipated via cooling, and the coolers needed are very expensive since they have to be powerful enough to dissipate all the concentrated hot. Both cooling and power consumption can be a quarter of datacenters day-to-day maintenance costs.

On the other hand, nowadays almost everyone has a mobile phone with a computer inside providing services like Internet and games among others, which is called to be a smartphone. Also have been appeared other devices like electronic books or tablets, etcetera. All of these devices have one essential and common characteristic: autonomy, they work without need to be power plugged. And this is achieved with batteries inside which work with energy, this is a basic physics principle where the batteries are charged with energy and then they gave the stored energy to the device, achieving the effect of device being power plugged.
So, as much energy the device consume, lesser the time battery will keep energy inside, and so less time of autonomy.

This means that in a mobile device, with a powerful computer (or microcomputer, since they do not provide all the characteristics of a typical computer) to avoid losing time of autonomy larger batteries are needed, and since a mobile device needs to be mobile, it needs to be soft, so it is needed constant research on batteries to make smaller electronic components and achieve more capacity with same or less weight. But at the same time batteries have to be cheaper enough so do not make devices too much expensive so anyone will buy it. The research is expensive in terms of money and time, and sooner technology won't be capable of offer that at required marketing times.

1.2 Saving energy

One way of solving both problems at once is making more energy saving computers. This is a current research area in several ways. For example, it has been researched how to save energy in green datacenters (equipped with solar collectors among others) scheduling jobs on predicted times where green energy will be available. Other way is an actual implemented design on almost all processors where they have several working frequencies, automatically adjusting to the lesser one when there are a few jobs on progress, or setting to higher ones when there are a lot of jobs running.

We are going to focus on other way of save energy, which it is found in the functional units of a microprocessor. The functional units of a microprocessor are the ones who execute code instructions, like adding two integers or comparing them, multiplying two real numbers, etcetera.

Depending on the functional unit an instruction consumes more or less energy. In terms of energy, it is not the same adding two integer numbers than adding two float integer numbers.

1.2.1 Vectorization

The most common and known functional unit of a micorprocessor is the ALU (Arithmetic-Logic Unit). It is implemented on all microprocessor and offers functionalities like adding two integer numbers, subtracting them, shifts a number, comparisons, etcetera. All program codes use these functionalities.

Sooner in time more calculation power were needed on functional units, since for applications which required vectorial processing, like many algorithms in research areas like physics or biologics or even in video-games. Vectorial processing means the calculations with vectors, which implies a lot of little additions, multiplications or others at a time. One single addition is fast, but 2.000 additions at a time implied 2.000 separated additions in terms of ALU, and this is slow when it is required to do it a lot of times.

To solve this problem in 1998 AMD introduced a new set of instructions called 3DNow!, these instructions were executed in a special functional unit. These functional unit allowed adding four integers at once. So, in fact, that is a vectorial functional unit, and these instructions were vectorial instructions.
The year later Intel also introduced they vectorial instructions set, SSE, which is actually is one of the most known instructions sets.

The implications of these sets were not only that applications worked almost 3 times faster, so improving performance, but also it had a energy saving side effect. These functional units consume more energy than ALU, but less energy than executing 4 instructions on ALU. So the programs using these instructions were actually saving energy and were faster than other ones.

Since then vectorial instructions sets had grown a lot, they admit all ALU functionalities, but also admit operations with float numbers and complex ones like adding some numbers and then, based on the zero flag (or other flags), jump to a line of code (which is called a branch instruction), among others.
Actually SSE version 5 is being developed and AMD have also developed a new versions of 3DNow! and is actually working on a new vectorial instruction set.

Nowadays some vectorial instructions set like SSE's are provided in almost all microprocessors.

1.2.2 Problem and compilers solution

These instructions, however, have a main drawback, which is compatibility between processors. With each new version of the SSE, for example, new vectorial instructions where added, but these instructions, obviously, are not supported on previous versions, so older processors can't run programs using these instructions.
Also there are compatibility problems between manufacturers. Not always a vectorial coded program for an AMD processor will work in an Intel one. In fact the instructions sets compatibility between processors manufacturers are of a few ones.

This is the main reason why the actual software usually is not implemented in a vectorial way, since it will not be portable. Portable means that the same code can work with other architecture or operating system without changing anything.
If a program is coded using, for example, SSE version 4, it will not be portable to computers with processors like the first Intel with SSE, unless the program code is changed to remove the missing instructions on SSE and transforming it to a simple scalar instructions.

This is not acceptable for many applications, so programs coded in a vectorial way are reduced to a few research areas where it is coded to a specific architecture.

As it has been said, the problem of not coding in a vectorial way is we are going to waste more energy and programs will take longer to execute, which gave us the two main problems described at the beginning.

But there is another way to try solving this problem. This is in a compiler. The compiler is the stage where a program code is transformed to an assembly code and later to binary code.
Since compilers parse all the code to achieve this transformation, they often can apply some optimizations on programs, and one of them is vectorizing some loops.

That is, let us imagine we have the following code:

```
for(int c = 0; c<1024; ++c) {
    C[i] = A[i] * B[i];
}
```

Code 1: Non-vectorial codification

Obviously this code is not coded in a vectorial way, since in every iteration we are doing a scalar multiplication. But actually it is fully vectorial since it can be vectorized with the following transformation:

```
for(int c = 0; c<1024; c+=4) {
    C[i:i+3] = A[i:i+3] * B[i:i+3];
}
```

Code 2: Vectorial codification

Where X[i:i+3] represents an array of four elements, which in hardware terms will be something like the following draw:

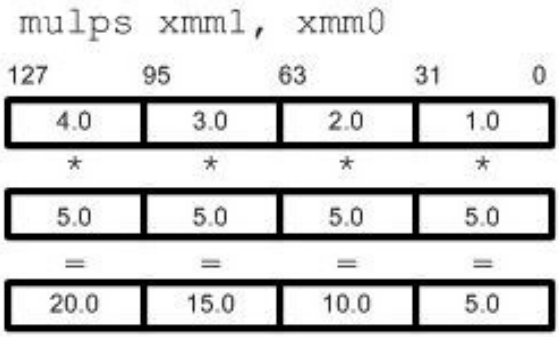


Figure 1: Vectorial multiplication

Each line is just a register of 128 bits, where each 32 bits are interpreted as a single number of our array.

The last register is where the results are stored, and they are calculated at the same time on the vectorial functional unit.

The compiler can achieve this code transformation if and only if the following conditions apply:

a) **The loop trip count must be known at entry of the loop at runtime.** So the end must be invariable, it cannot be data dependent.

   This restriction is because when we vectorize a loop, the number of iterations is decreased (in that case, divided by 4, the number of multiplications at the same time we can do).

   If compiler cannot know the number of iterations we are going to do for any reason, it cannot change the number of iterations properly to avoid doing more (or less) multiplications than code wants to do, so actually the compiler would be changing the code semantics.

b) **Single entry and single exit**. That is, there can not be any case where the loop exits before the invariant end, and also the loop can not start in different position counts.

   Actually this is implied by first condition, and the reason is the same one. If suddenly the loop exits before completing all iterations, it could have done some extra operation (or have left some to do).

c) **Straight-line code**. Each iteration must do the same calculations, which implies there can not be any branches, only the “if” branches are allowed if and only if they can be masked into simple assignments, which usually can be done.

   This is because if in some iteration we do an addition, and in the following one a multiplication, then we cannot just do four additions at the same time, since only two of them are really done by the code.

So the only exception to the rule is when the branches or conditions can be calculated. For example, let's assume an if where it checks if some number is odd or even. This branch can be easily calculated with module operation, so in this case the branch could be removed by compiler (since this is a detection compilers can do), and replaced properly by module operations.

Then the loop is just straight-line and all the operations are always the same, so it can be possible to vectorize the loop (if the rest of conditions are fulfilled).

d) **The innermost loop of a nest.** Except if it is an original outer loop and has been transformed into an innermost loop because of a previous optimization phase, in which case under the vectorization optimization phase point of view will be just an innermost loop fulfilling the condition.

It is not possible to vectorize operations in a loop which at a time contains another loop, since it is not known if the inner loop is going to change the results of the data used in the outer most loop. Compilers cannot know this because it would break polynomial time limit.

e) **No function calls**.

Basically is the same reason as previous one.
A function call analysis cannot be done each time compiler detects a loop to try to vectorize it since it would break, once again, the polynomial time limit of compiling.

f) **All memory accesses must be contiguous**.

This is due compilers cannot calculate memory addresses, and so they cannot check contents of arrays or other structures. So only if accesses are contiguous compilers can be sure that data won't be changed between iterations and so that there are no data dependencies between them (see next condition).

g) **There can not be data dependencies between iterations**.

Let's take the following loop example (in C):

```
for(int c = 1; c<9; ++c) {
     A[i] = A[i-1] + B[i];
}
```

Code 3: Non-vectorial code

Here we can see that in second iteration will be using data calculated in previous iteration. So we cannot do the two additions at the same time since the second addition needs the result of first one. This is a data dependency, and this avoids us to vectorize the code.
Not knowing about data dependencies between iterations is a very common problem and prevents compilers to automatically vectorize codes.

1.3 Motivation

Previous restrictions can only be accomplished by simple codes, but often the codes require some nested loops, functions and several other complex codes. In this cases the codes can usually be partial vectorized, or even fully vectorized but compilers can not manage that and do not understand it.

For example, the following code (converts a RGB color to a YUV color) can not be transformed by compilers:

```
float* yu;
float* rg;
len*=12;
for(j = 0; j<N_iter; j++) {
  yu = &(yuv[0].y);
  rg = &(rgb[0].r);
  for(i = 0; i<len; i+=12) {
    *yu++ = 0.299*(*rg)+0.587*(*(rg+1))+0.114*(*(rg+2));
    *yu++ = 0.436*(*rg)-0.147*(*(rg+1))-0.289*(*(rg+2));
    *yu++ = 0.614*(*rg)-0.515*(*(rg+1))-0.099*(*(rg+2));
    rg+=3;
  }
}
```

Code 4: Non-vectorizable code by compilers

Because of the pointers used, compiler cannot be sure of data dependencies due to not contiguous memory positions (condition f). But an human can do something like this:

```
    __m128 pa,pb,pc;
      pa = _mm_setr_ps(0.299,0.587,0.114,0.299);
      pb = _mm_setr_ps(0.436,0.147,0.289,0.436);
      pc = _mm_setr_ps(0.614,0.515,0.099,0.614);


     float* yu;
     float* rg;
     struct resultats {
       float total[9][4];
     }valors;

     len*=12;
     for(j = 0; j<N_iter; j++) {
       yu = &(yuv[0].y);
       rg = &(rgb[0].r);
       for(i = 0; i<len; i+=48) {
         __m128 *op1;
         op1 = (__m128*)rg;

         __m128* res = (__m128*)&(valors.total[0][0]);
         *res = _mm_mul_ps(*op1,pa);
         res = (__m128*)&(valors.total[1][0]);
         *res = _mm_mul_ps(*op1,pb);
         res = (__m128*)&(valors.total[2][0]);
         *res = _mm_mul_ps(*op1,pc);

         rg+=4;
         op1 = (__m128*)rg;
         res = (__m128*)&(valors.total[3][0]);
         *res = _mm_mul_ps(*op1,pa);
         res = (__m128*)&(valors.total[4][0]);
         *res = _mm_mul_ps(*op1,pb);
         res = (__m128*)&(valors.total[5][0]);
         *res = _mm_mul_ps(*op1,pc);

         rg+=4;
         op1 = (__m128*)rg;
         res = (__m128*)&(valors.total[6][0]);
         *res = _mm_mul_ps(*op1,pa);
         res = (__m128*)&(valors.total[7][0]);
         *res = _mm_mul_ps(*op1,pb);
         res = (__m128*)&(valors.total[8][0]);
         *res = _mm_mul_ps(*op1,pc);

         *yu++=valors.total[0][0]+valors.total[0][1]+valors.total[0][2];
         *yu++=valors.total[1][0]+valors.total[1][1]+valors.total[1][2];
         *yu++=valors.total[2][0]+valors.total[2][1]+valors.total[2][2];

         *yu++=valors.total[0][3]+valors.total[3][0]+valors.total[3][1];
         *yu++=valors.total[1][3]+valors.total[4][0]+valors.total[4][1];
         *yu++=valors.total[2][3]+valors.total[5][0]+valors.total[5][1];

         *yu++=valors.total[3][2]+valors.total[3][3]+valors.total[6][0];
         *yu++=valors.total[4][2]+valors.total[4][3]+valors.total[7][0];
         *yu++=valors.total[5][2]+valors.total[5][3]+valors.total[8][0];

         *yu++=valors.total[6][1]+valors.total[6][2]+valors.total[6][3];
         *yu++=valors.total[7][1]+valors.total[7][2]+valors.total[7][3];
         *yu++=valors.total[8][1]+valors.total[8][2]+valors.total[8][3];
       }
}
```

Code 5: Human-vectorized code

The instructions and types starting by “_” are instructions and types of SSE instruction set library.
So the code was almost fully vectorial but the compiler couldn't detect it since it could not know the memory position of the rg pointer, and so it cannot detect if there were or not any memory dependencies. Since if there is a memory dependency the code is not vectorial, the compiler cannot take that risk because it could change the semantic of program.

But what if we could know at every moment which dependencies exist between instructions? In that case we could write better techniques to transform more codes into vectorial codes. The compilers can only know the dependencies and do better transformations only breaking the polynomial time limit, which isn't acceptable for a compiler.

Processor can know dependencies in linear time, since the processor is actually executing code so calculating the addresses and accessing memory.
Briefly, our proposal solution is to change the design of a microprocessor to dynamically transform the code. We are going to first detect several executions of the same instruction, then place their operands in a vectorial way, like one shown in the previous draw, and if there are no memory dependencies between four repetitions of instruction we are going to generate a vectorial instruction replacing the scalar ones. This will be extensively explained in chapters 2 and 3.

This way we are solving the main problem of compiler. Also we are solving portability of programs since no recode is needed at all, the same binaries are valid. And also we can vectorize some old codes that where not optimized at all.

1.3.1 Goal

Our main goal here is to save energy at cost of performance. Performance will be probably lost since the instructions are stalled until four instances of them are found, which implies that they are going to be executed later than originally, and so programs will finish later.
But we hope not to loss too much performance since we expect many of these blocks to be converted to vectorial instructions, and so the vectorial instruction will finish not too much later than the last of the group individually would. Also this means that we really expect to improve enough the energy consumption because we will be reducing from four original instructions to one single vectorial instruction, and it has been explained that energy consumption of vectorial functional units is lesser than executing four scalar operations.

2. Computer architecture overview

A computer is a general purpose device that can be programmed to carry out a set of arithmetic or logical operations. Since a sequence of operations can be readily changed, the computer can solve more than one kind of problem.

Conventionally, a computer consists of at least one processing element, typically a Central Processing Unit (CPU) and some form of memory. The processing element carries out arithmetic and logic operations, and a sequencing and control unit that can change the order of operations based on stored information. This element is also commonly referred to as the processor (or microprocessor).

Also a computer has several peripheral devices with several function such as allowing information to be retrieved from an external source, and the result of operations saved and retrieved (a hard disk). Other common peripherals are the screen monitor where we can interact with a computer in a visual manner, and keyboard which will allow us to give instructions to computer.

All of this three elements (CPU, memory and I/O devices) consists in the basic elements of a computer. The way the elements interact is defined as a computer architecture.

2.1 Von Neumann architecture

There are several computer architectures, but we are going to focus with the most common one, the Von Neumann architecture, which we represent in the following block diagram:

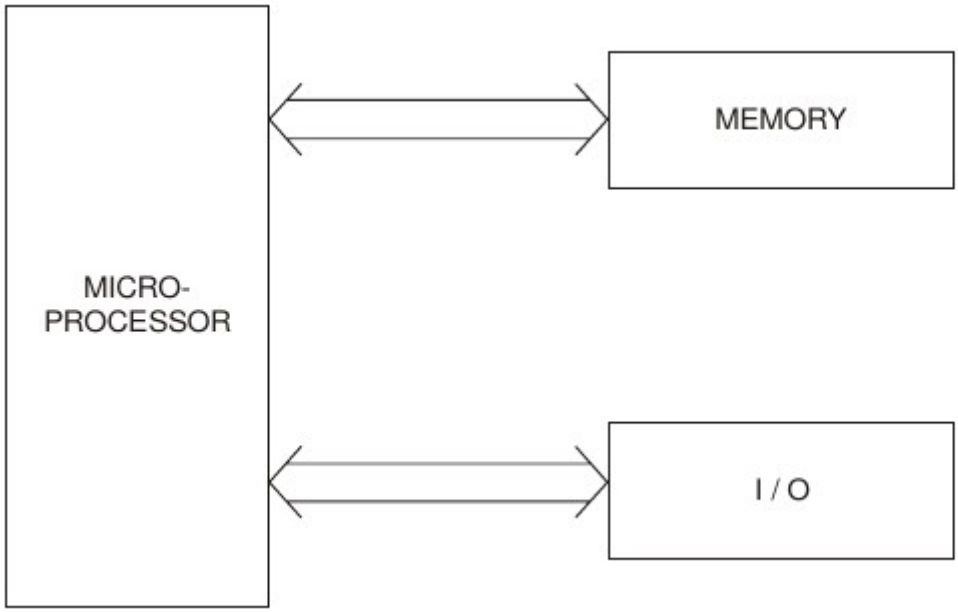


Figure 2: Von Neumann architecture

Nowadays this schema has evolved a lot and we have several subdivisions in each block, but essentially the architecture is the same. Briefly the tasks of each block are the followings.

- **Memory**: it stores the programs code (i.e instructions) and the stored data.

- **I/O**: it stands for input / output and it refers to the peripherals connected to computer. For example, a keyboard and mouse, a printer, an screen, and any other devices belong to I/O block.

- **Microprocessor**: that's the basic operational unit of a computer. It gets the program code from memory and executes it, performing any I/O tasks required and storing or retrieving any memory data.

2.2 Processor design overview

Our project focuses on changing the processor design. The microprocessor or CPU reads each instruction from the memory, decodes it and executes it. It processes the data as required in the instructions. The processing is in the form of arithmetic and logical operations. The data is retrieved from memory or taken from an input device and the result of processing is stored in the memory or delivered to an appropriate output device, all as per the instructions.
To perform all these functions, the microprocessor incorporates various functional units in an appropriate manner. Such an internal structure or organizational structure of microprocessor, which determines how it operates, is known as its architecture.

A typical microprocessor architecture is as follows:

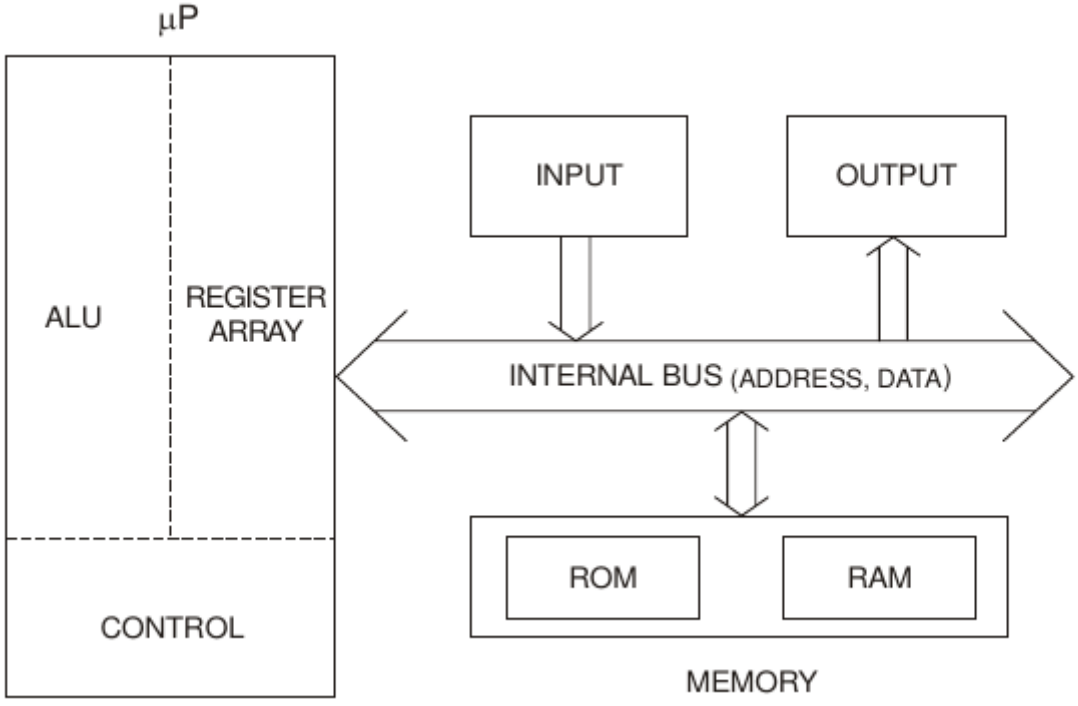


Figure 3: Microprocessor architecture

In the picture only an ALU functional unit is shown, but actually there are several other units like a vectorial functional unit (commonly called as SIMD unit, where SIMD stands for single input multiple data), a float operations unit, multiplication and division units,

etcetera.

Briefly, each block has the following functions:

- **Register file** (register array in figure): this is a very important part of a microprocessor. A register is a kind of short term data memory. Physically getting data from and to memory is a slow operation in comparison of execution speed of a single instruction within processor, i.e, is a processor bottleneck. To solve this problem registers were introduced as a faster memory, but a little one. So when an instruction needs data memory to operate with, first the data is stored into registers and then processor just operates with them. Once the operation is complete results are write back into memory and registers can be reused by other instructions.
  Register file is in charge of manage all the processor registers, retrieving and storing data from and to it.

- **Control**: gets instructions from memory, decodes them, gets the source registers data from register file and pass it to the functional unit. Once the functional unit executed the instruction, it returns to the control unit where it stores the data into the destination register or to memory.

- **Functional units**: they execute the instructions itself, adding, dividing, comparing or whatever the instruction do. They do not store data on register nor memory since this is the purpose of the control unit.

2.2.1 Pipeline of a processor

The control unit as it has been seen is in charge of lots of work. For this reason it is also divided in several areas itself. Putting all the units and subdivisions together we get the following five stages of first processors, commonly called the processor pipeline:

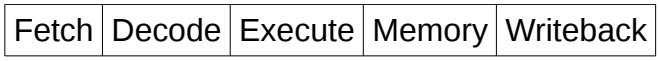


Figure 4: Basic 5-stage pipeline

Where each stage is in charge of the following:

- **Fetch**: here the processor search in memory for the next instruction based on PC.

- **Decode**: once instruction has been fetched, we have to decode the bits to know which instruction is (an add, a multiplication, a branch, etcetera), how many source register it uses (if any), destination registers, etcetera. It passes this information to next stages in order to know what to do.

- **Execute**: we can say that every functional unit is an execution stage. When an instruction is executing on a functional unit, we say that it is on execution stage. So the set of functional units compound the execute stage of a processor.

- **Memory**: in this stage data is stored in memory and we get data from memory if required.

- **Writeback**: this stage writes the results on destination registers.

2.2.2 Linear processors

The execution of instructions were like in the following draw:

Figure 5: Linear processors

These processors are called linear ones because each instruction have to wait the previous one to complete all stages in order to start it's execution. These processors usually are also called subscalar CPUs.

2.2.3 Scalar processors

The linear scheme is suboptimal because when an instruction completed an stage, this stage is no longer used until next instruction. So it have to wait the whole 4 cycles to work again. For this reason a new way of executing instructions were introduced where each cycle an stage is executing one instruction, so all the stages are always working. This is called an scalar processor and it works as the diagram below:

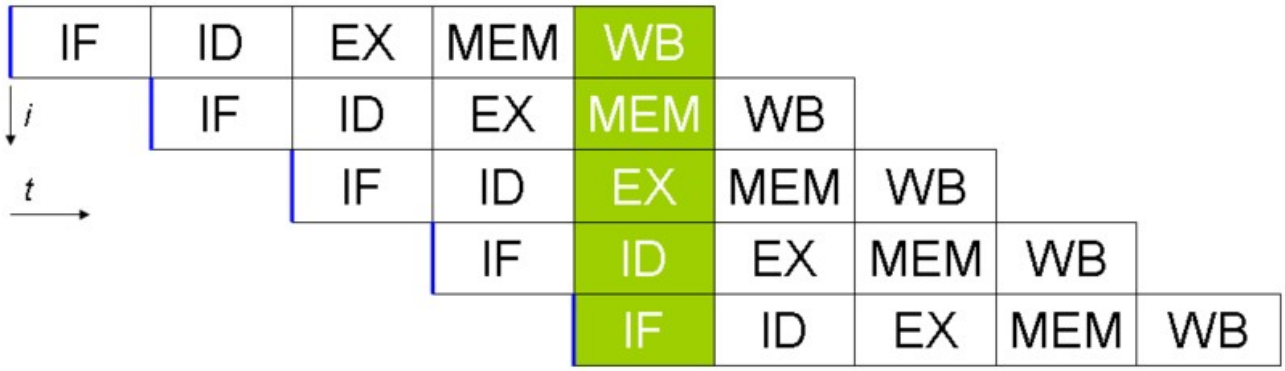


Figure 6: Scalar processor

2.2.4 Multi-cycle processors

On our figures we can show an execute stage that takes only one cycle to complete (each block represents a cycle to complete). But as we said there are several functional units who executes instructions in that stage, and it is common that not all takes the same time to complete.
Also it is common that memory stage takes different number of cycles depending on if it is loading data or saving it.

When it happens that stages can take different amount of cycles depending of what we are doing it is said to be a multi-cycle processor. In our case stages until execute will only take

one cycle, so only stages since execute or after that would take more than one cycle, which is a very common design. Otherwise it is harder to get rid of some data hazards (we will talk about hazards on 2.3).

Here is an illustrative example:

	Clock Number										
	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	D	M1	M2	M3	M4	M5	M6	M7	M	WB
...		F	D	E	M	W					
...			F	D	E	M	W				
ADDD F2, F4, F6				F	D	A1	A2	A3	A4	M	WB
...					F	D	E	M	W		
...						F	D	E	M	W	
LD F2, 0(R2)							F	D	E	M	WB

Figure 7: Multi-cycle processor

2.2.5 Superscalar processors

The last kind of pipeline design we are going to analyze is the superscalar one. This kind is exactly like scalar pipeline, except that now an stage is processing several instructions at the same time. Actually it is a pipeline replication, and each pipeline is called a thread. The following figure illustrates it:

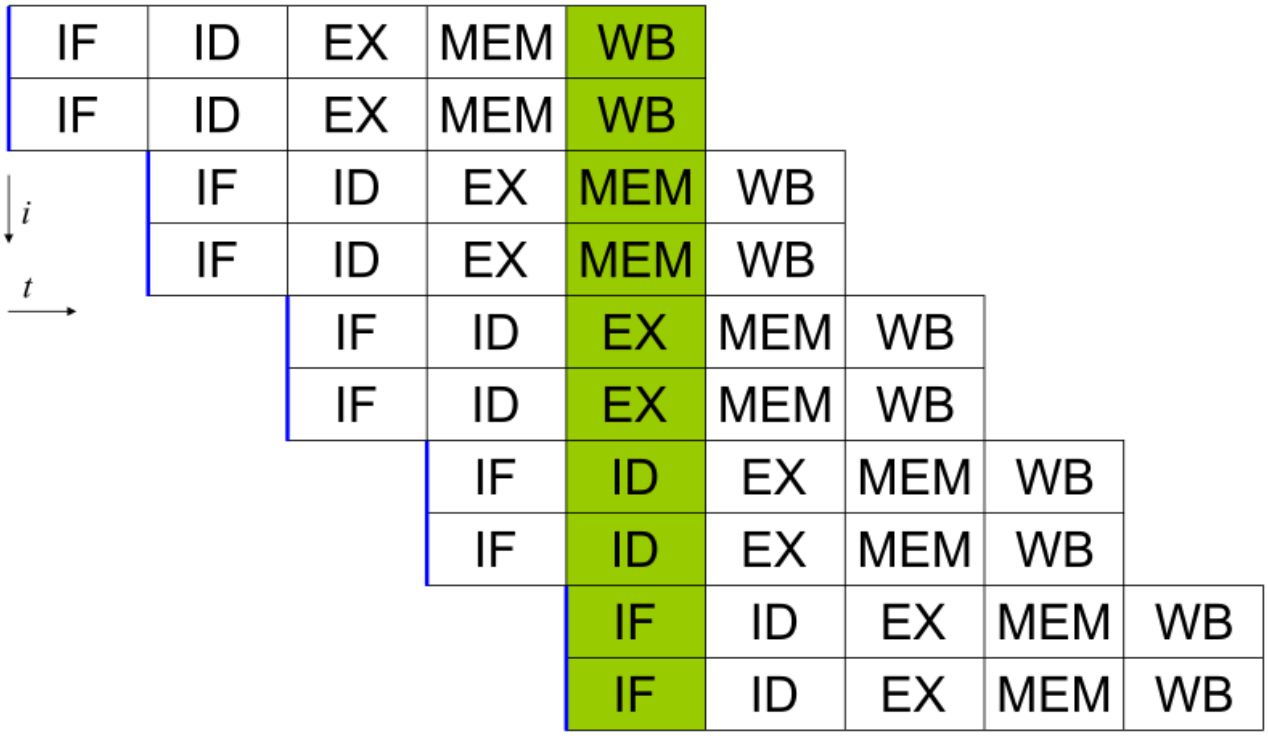


Figure 8: Superscalar processor

As it happens with scalar processor, we can have a superscalar multi-cycle processor. Indeed most actual processors are.

2.2.6 CPI

To measure the performance of both models we have to introduce a measurement unit. This unit will be the cycles per instruction (CPI). That is, how much cycles takes for an instruction to complete?

Now we can measure the performance of both models. In the linear one it is clear that an instructions takes all the five stages to complete, and since an stage takes one cycle, the CPI of the linear processor is of 5 cycles.
In the scalar processor each cycle we complete an instruction, so we can say that in best case scenario the CPI is of 1 cycle.
So the scalar processor if 5 times faster than linear one.

In comparison with superscalar processor, assuming the two threads shown in figure 8, CPI is of 0.5 cycles. So it is 2 times faster than scalar processor.

2.3 Hazards

The previous described scalar pipeline (and their evolutions) has several kinds of problems. Since in a given instant several instructions are in execution on the pipeline, there may be some data problems which may break the program semantics. A processor must maintain and guarantee program semantics, so it must take actions on these threats.

There are three kinds of hazards:

- **Data hazards**: these hazards occurs when an instruction may be reading or writing different data from a register or memory than it would be if it was a linear execution.

- **Structural hazards**: these hazards can't occur on our described models. They only can occur on processor pipeline designs where the stages are not always executed by all instructions, that is when two different instructions can pass through different stages and not through all of them. This kind of designs are considered as non-linear, which are not of our interest since almost all current processors are only based on linear designs.

- **Control hazards**: these hazards occurs because of branches and function calls. A program code may have some execution order breaks, that is, they can jump from instruction eleventh to instruction thirty first. On our pipeline each cycle we are retrieving the next instruction, but maybe some of older instructions is a branch one and we must take that jump, so we are going to execute some instructions that should not be executed. This is a control hazard and instructions not supposed to be executed must be removed from pipeline.

2.3.1 Data hazards

In the following lines we are going to describe the three kinds of existing data hazards. For all of them we are assuming a scalar processor, but it can be applied to any of their evolved variants, unless it is explicitly said which kind of pipeline applies.
In linear pipeline no data hazard can exist at all since there is no more than one instruction at a time.

- **Read after write**: this hazard occurs when one instruction writes a data and the next one reads and use it. The second one cannot read the data until the first one has write it, so processor must ensure that data is written onto register prior to execute the second instruction. Example code:

```
i1. R2 ← R1 + R3
i2. R4 ← R2 + R3
```

Code 6: Read after write data hazard

When instruction 1 is on memory stage, instruction 2 will already be on execute stage. So processor will be doing an addition with invalid data since register 2 still do not have the correct data on it.

- **Write after read**: this hazard can only occur if there is a chance that a second instruction writes data before a previous one read it. In that case, we can have a code like this:

i1. R2 ← R1 * **R3**
i2. **R3** ← R5 + R0

Code 7: Write after read data hazard

If instruction 2 could write the R3 register before instruction 1 reads it, then instruction 1 will be multiplying incorrect data. Only on superscalar processors it can happens if instructions are on different threads, and the first one is blocked because of some data hazard or any other reason.

- **Write after write**: it can happens on any multi-cycle processors. Let's take the following instructions as an example:

```
i1. R2 ←  R4 * R7
i2. R2 ←  R1 + R3
```

Code 8: Write after write data hazard

- Assuming that ALU unit takes 1 cycle to complete and multiplication unit takes 4, instruction 2 will complete before instruction 1 does, and so could write register 2 before the first one does. Then, when instruction 1 would complete it would rewrite register 2 with invalid data.

2.3.2 Eliminating hazards

Read after write is commonly resolved adding what it is called a pipeline bubble, keeping the execution of instructions on their current stages until the hazard disappears. Detection of hazard is done checking that for each source register any older instruction on pipeline is going to write it. If this situation occurs then the bubble is inserted. In the following draw we can see a pipeline with bubble:

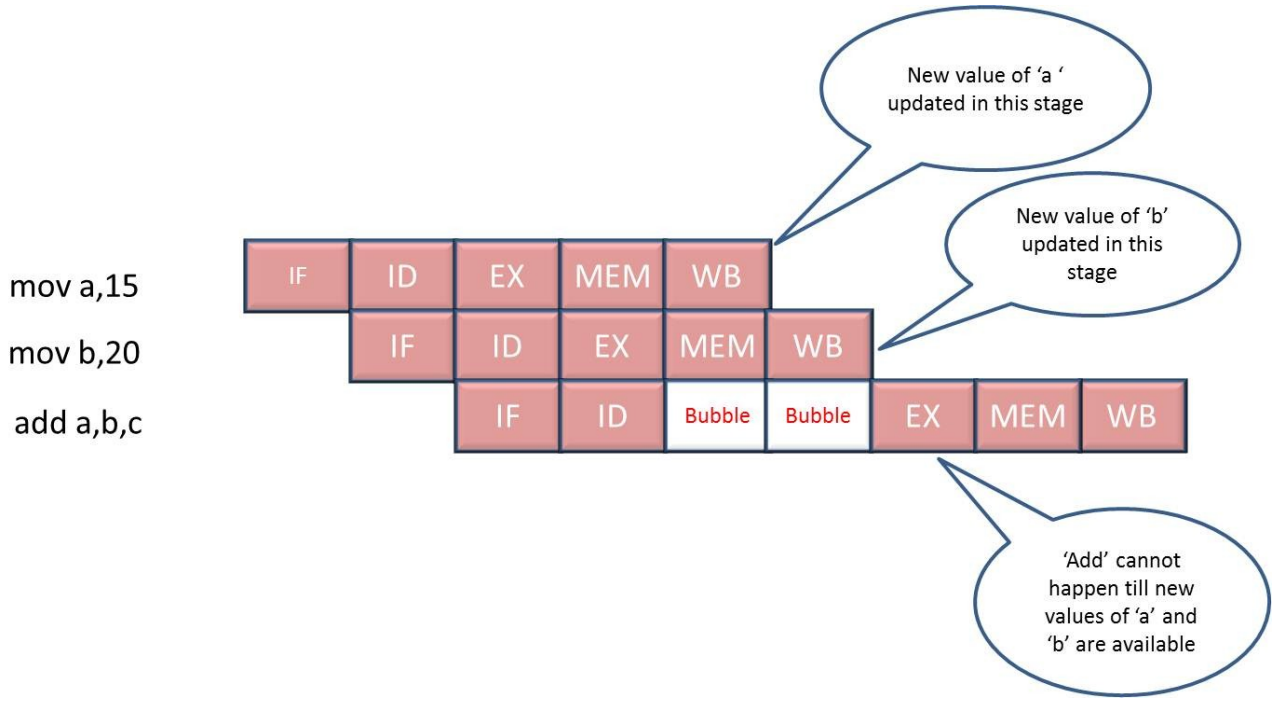


Figure 9: Pipeline bubble

2.3.2.1 Rename stage

With processors engineering evolution it appeared multi-cycle processors and so the write after write hazard. To eliminate it a new technique was created, called register renaming. This usually is presented as a separate stage on modern processors, but on our pipelines we have seen it placed on decode stage.

The idea of rename stage is to rename destination registers of instructions, so the two instructions that were writing to the same register now are writing to different ones, and so the hazard disappears. There are many renaming algorithms applying this concept, but we are going to review a simple one just to get an idea of how to resolve the hazard.

Here we have to introduce the concept of physical register and logical register. The processor receives from instructions decoding a register numbers that are not intended to match exact real registers on register file. These numbers are only references of program data, so processor only must ensure that it keeps semantic meaning of these registers. They are called logical registers.
The physical registers are those that are on the register file.

So when a new instruction arrives processor assigns a logical register to a physical one. To do this processor keeps an structure of free registers, which are registers not used by any other instruction on pipeline, and it assigns new physical registers from this pool.

Once a register is not longer used by any other instruction, it is put again on the free registers pool.

On the other hand, for logical source registers of an instruction processor searches the assigned physical register to each logical register, and so replaces it by the physical ones. This way source registers are now referencing the real physical registers that will be written.

With this register renaming it can be seen that write after read hazard is also eliminated. Because when the second instruction arrives it will be assigned a free physical register, so the source register of the first one is no longer matching the destination register of the second.

About the multi-cycle processors lasts a consideration, the functional units that takes more than one cycle to complete are usually pipelined too. This allows processors to reduce CPI.

To take advantage of that fact processors introduced a new stage before execution stage, called issue stage, which is in charge of manage functional units and blocks instructions whom their functional units have no free slots (they cannot allow more instructions), or to *issue* the instructions whom functional unit have some free slot.

2.4 Final pipeline

Once we have defined the hazards and the elimination procedures, we have now the following eight common stages of a pipeline:

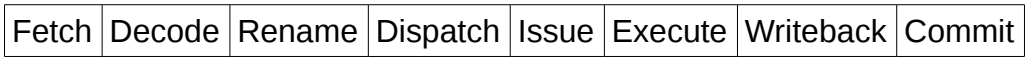


Figure 10: Basic stages of current processors

The stages we haven't described yet are the following:

- **Rename**: this stage is in charge to assign source and destination registers to physical registers as described previously.

- **Dispatch**: in this stage the processor manages the read after write hazards between instructions and blocks instructions (introducing pipeline bubbles) until their source registers are ready, so they have the correct data. Once they are ready the instruction is passed to issue stage.

- **Issue**: manages the functional units as described previously.

- **Commit**: this stage recovers from control hazards. Usually we predict to fetch the next instruction, but maybe we have a branch which creates a control hazard. In this case the prediction will have failed and in our pipeline we'll have instructions that should be removed. Here is when commit stage comes in action. When an instruction should be removed but it finished (executed and wrote back results), this stage is in charge on doing the “reparations” needed to keep the semantic of programs and finally removing the instruction.
  So when an instruction was correctly predicted, this stage also serves as a

confirmation unit.

- **Writeback**: here we have a difference with previous described pipelines. While in the previous pipelines we had a memory stage, in charge of operations to and from memory, and a writeback stage in charge of storing data to registers, here we put these two stages together in a single writeback stage. Division between memory and register file operations is a common division in processors, but as they have the same purpose we say that they are the same block.

Current processors have more stages, some recent processors have 23 or more stages. But they are only pipelines inside described stages.

3. Design

In loops often there are arithmetic instructions repeated several times, and the iterations of these instructions are data-independent so it is possible to vectorize it. If we make one vectorial instruction of four scalar instructions, we will be saving energy. Our design is based on this idea.

In our design we are going to change the rename stage of processor. In the following lines we describe some existent rename algorithms.

3.1 Rename algorithms

As it was mentioned in previous chapter, renaming consist of changing destination register names of instructions to avoid to instructions writing on the same register producing a data hazard, which will stall our processor's pipeline.

3.1.1 Scoreboard

Scoreboard is not an actual rename algorithm, but it is the immediate predecessor of rename algorithms as it was also introduced to avoid write after write and write after read data hazards. It was developed in 1966 and it was later used in processors like Intel Pentium.
Nowadays it's basic table structure (simplified) it's still used to manage ready source register of instructions, and so we do on our proposal design.

Its basic idea is to keep information about register to be read and written to manage hazards.

This algorithm have four stages:

- **Issue**: here this stage is slightly different as described previously. In this stage processor keeps information about which are source registers which will be read and which is the destination register to be write. It keeps this information onto a table, called the *scoreboard* table giving the algorithms name. If destination register is waiting to be write for a previous instruction, then instruction is stalled until previous instruction completes and writes the register and this way write after write hazard is resolved. Also an instruction could be stalled if his functional unit is busy. In the following figure an schematic scoreboard is shown:

		dest	*S1*	*S2*	*FU for j*	*FU for k*	*Fj?*	*Fk?*
Busy	*Op*	*Fi*	*Fj*	*Fk*	*Qj*	*Qk*	*Rj*	*Rk*
No								
No								
No								
No								
No								

Figure 11: Scoreboard structure

- **Read operands**: once instruction has been issued to the functional unit, it gets stalled until all his source operands are ready. An operand is only considered to be ready until producers instructions really writes registers. This avoids read after write data hazards.

- **Execution**: when all operands have been fetched, the functional unit starts its execution. After the result is ready, the scoreboard is notified.

- **Writeback**: in this stage instructions are about to write destination registers. But this is not done until all previous instructions reading the operand complete their *read operands* stage. This way write after read data hazard is addressed.

The table shown in figure 11 represents a row for each functional unit we have. And rows information is the following:
Busy bit: this bit indicates if functional unit is busy or not. That is, if some instruction is using it.
Op: this indicates which operation instruction is being executed on functional unit. For instance, in an ALU unit it could be an addition, a shift, a subtraction, etcetera.
Dest Fi: indicates which register is going to be written.
S1 (Fj): indicates first source register.
S2 (Fk): indicates second source register.
FU for j (Qj): indicates which functional unit produces first source register.
FU for k (Qk): indicates which functional unit produces second source register.
Fj? (Rj): indicates if first source register is ready (already written by producer instruction).
Fk? (Rk): indicates if second source register is ready.

For each instruction, the issue stage checks if his functional unit is ready (busy bit is set to 0), and his destination register is checked on scoreboard. If register it's about to be written by some instruction (it appears in a *“Dest Fi”* column with busy bit of functional unit set to 1), then instruction is stalled since this is a write after write data hazard.
Once instruction can issue scoreboard table is updated with his data.

The pseudocode for issue stage looks as follows:

```
  function issue(op, dst, src1, src2)
     wait until (!Busy[FU] AND !Result[dst]); // FU can be any functional unit
that can execute operation op
     Busy[FU] ← Yes;
     Op[FU] ← op;
     F_i[FU] ← dst;
     F_j[FU] ← src1;
     F_k[FU] ← src2;
     Q_j[FU] ← Result[src1];
     Q_k[FU] ← Result[src2];
     R_j[FU] ← not Q_j;
     R_k[FU] ← not Q_k;
     Result[dst] ← FU;
```

Code 9: Scoreboard's issue stage

Stalling at issue stage implies that no other instruction will proceed, otherwise we could have some unmanaged hazards because scoreboard data is not updated until instructions can issue.

Once instructions issued, it passes to read operands stage where source registers are read. To read a source register we must wait they are ready, indicated by bits *"Fj?"* and *"Fk?"* (when they are set to 1, they are ready) meaning that producing instructions completed and writted results in registers. This way we avoid read after write data hazard. To do so scoreboard needs to know which functional unit produces each registers in order to update the ready bits. This is maintained with the following structure, where for each register we write the producer functional unit:

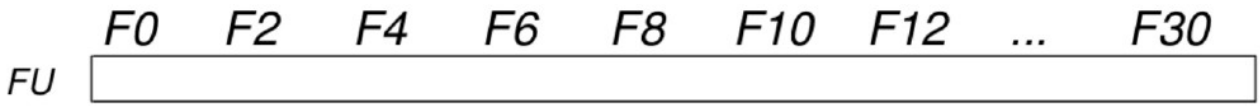


Figure 12: Producer functional unit table

Pseudocode for read operands stage:

```
  function read_operands(FU)
     wait until (R_j[FU] AND R_k[FU]);
     R_j[FU] ← No;
     R_k[FU] ← No;
```

Code 10: Scoreboard's read operands stage

Once registers are ready, instruction passes to execution stage where it is really executed. Once result is ready, so execution completed, scoreboard is notified and it gets passed to the write result stage.

So execution stage pseudocode is as simple as follows:

```
function execute(FU)
   // Execute whatever FU must do
```

Code 11: Scoreboard's execution stage

In the last stage scoreboard checks that no other instruction is reading our destination register, to avoid write after read data hazard, and if satisfied then scoreboard is updated setting bits to 1 in *"Fj?"* and *"Fk?"* columns properly.
If some instruction is waiting to read the register, instruction once again gets stalled until register is read and it can write result.

Pseudocode writeback stage:

```
function write_back(FU)
   wait until (∀f {(Fj[f]≠Fi[FU] OR Rj[f]=No) AND (Fk[f]≠Fi[FU] OR
Rk[f]=No)})
    foreach f do
        if Qj[f]=FU then Rj[f] ← Yes;
        if Qk[f]=FU then Rk[f] ← Yes;
    Result[Fi[FU]] ← 0;
    Busy[FU] ← No;
```

Code 12: Scoreboard's writeback stage

The main drawback with described algorithm is that whenever a hazard could exist it stalls instructions limiting too much processors performance. In order to solve this problems appeared the rename algorithms, and one of them is tomasulo algorithm.

3.1.2 Tomasulo algorithm

This is the first rename algorithm, developed at 1967 by Robert Tomasulo for IBM. This algorithm allows out-of-order instruction execution and avoids stalling instructions due to write after write data hazards, which improves significantly processors performance.

The basic idea behind tomasulo – and all rename algorithms - is to rename register destination registers to other unused registers, avoiding write after write hazard because two instructions previously writing on the same register will now write on different registers, so resolving hazard.

To better explain this algorithm we have to introduce a new concept, the reservation stations. A reservation station consists of a slot to allocate an instruction previous to enter his functional unit pipeline.
Each functional unit has several reservation stations where it can put instructions waiting to be executed while other instructions are being executed on pipeline, this way it is not necessary to stall all processors pipeline and so instructions using other functional units can proceed normally (if there isn't any hazard in-between).

There are two cases for an instruction to be kept on functional unit reservation station:

1. Functional unit pipeline is busy.
2. Instruction source registers are not ready (so it's waiting them to be ready).

Notice that with this schema it could happen that an instruction was kept in a reservation station forever even though his registers were ready, since it could happen that when his source registers get ready there were other instructions inside the pipeline and at the same time new instructions were arriving and got ready too, if the first instruction is unlucky this new instructions will go through pipeline before. So each functional unit have to maintain some kind of policy to avoid starvation of instructions and select instructions to execute in a smart way, but we are not going to focus on these polices because it is not the subject of our proposal design.

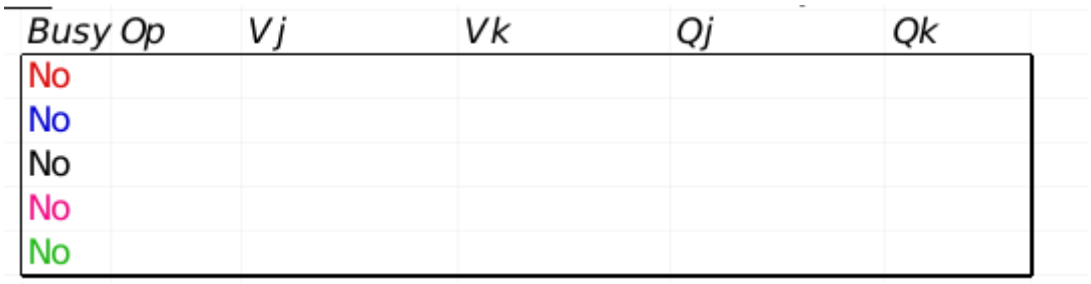


Figure 13: Reservation stations table

Reservation stations also store information about operation source registers, to easily find source registers state and values. The reservation stations look as the following figure:
As we can see it's design is similar to scoreboard table. In this case each row represents a reservation station instead of a functional unit, so while in scoreboard for ALU we only had one row, here we could have three rows for ALU if it reservation station had three slots.
The descriptions of the bits are the followings:
Busy bit: this bit indicates if reservation station is busy or not. That is, if some instruction is on it.
Op: this indicates which operation instruction is being executed on functional unit. For instance, in an ALU unit it could be an addition, a shift, a subtraction, etcetera.
S1 (Vj): indicates first source register.
S2 (Vk): indicates second source register.
FU for j (Qj): indicates which functional unit produces first source register.
FU for k (Qk): indicates which functional unit produces second source register.

Also we have the following to bits, which are not in the reservation station itself but in wires from last stages of functional units to issue stage which are wearing the calculated register names and it's values. This technique is known as short-path and it is useful to awake instructions before its producers fully complete (we don't have to wait until writeback stage to get his values). The bits are the followings:
Fj? (Rj): indicates if first source register is ready (already written by producer instruction).
Fk? (Rk): indicates if second source register is ready.

Reservation stations is a key concept in tomasulo algorithm since it is the essential part of his register renaming.

Tomasulo is compound by several stages, which functionality is as follows:

- **Issue**: if there is a free reservation station (there's no structural hazard), issue instruction and rename registers. Registers are renamed according to reservation station name. Names are as follows: RS*FUi* where FU is the functional unit name, for example RSALU, and *i* is the index reservation station of the functional unit. RS stands for “reservation station”. So if we place our instruction to first reservation station of the ALU, his destination register will be renamed to RSALU0.

- **Execution**: when source registers of instruction are ready and there isn't any other instruction already executing on functional unit, execute instruction. Otherwise wait until two conditions are true.

- **Writeback**: once instruction completes, write results and transmit it to all awaiting units. Also used reservation station is freed.

Since we are renaming destination registers on issue stage, we need to keep track of these renames in order to also rename the further consumer instructions. That is, let's imagine we have the following instructions:

i1. ADD R1, R0, R4
i2. ADD R3, R1, R2

Issue stage will rename the i1 R1 register to RSALU0, but i2 is reading this register. So it also must be renamed to RSALU0 achieving the following result:

i1. ADD RSALU0, R0, R4
i2. ADD R3, RSALU0, R2

So tomasulo keeps a table associating the logical registers to actually renamed registers. In our example will look as follows:

Logical Register	**Register renaming**
R1	RSALU0

Figure 14: Register rename table

Issue looks for source registers in the table and renames them with the associated names.

Altogether can be noticed that scoreboard and tomasulo stages are very similar, indeed its operation stages are almost the sames. The main difference is that reservation stations allows us to rename destination registers and so avoid two instructions writing on the same register. This little difference makes huge improvements on processors performance, since we don't need stalling all processor pipelines anymore.

3.1.3 Renaming through the reorder buffer

Another way of rename is through a reorder buffer (ROB). A ROB it's an structure which stores the values produced by noncommitted instructions whereas the architectural register file stores values of the last committed instruction. When an instruction ends its execution stage, its value is written into ROB, and when it commits this value is copied to architectural register file.

For each logical register we have an structure with a bit indicating if its latest value is in the register file or in the ROB. In the last case we also store a pointer to the ROB value position. The scheme is shown in the following figure:

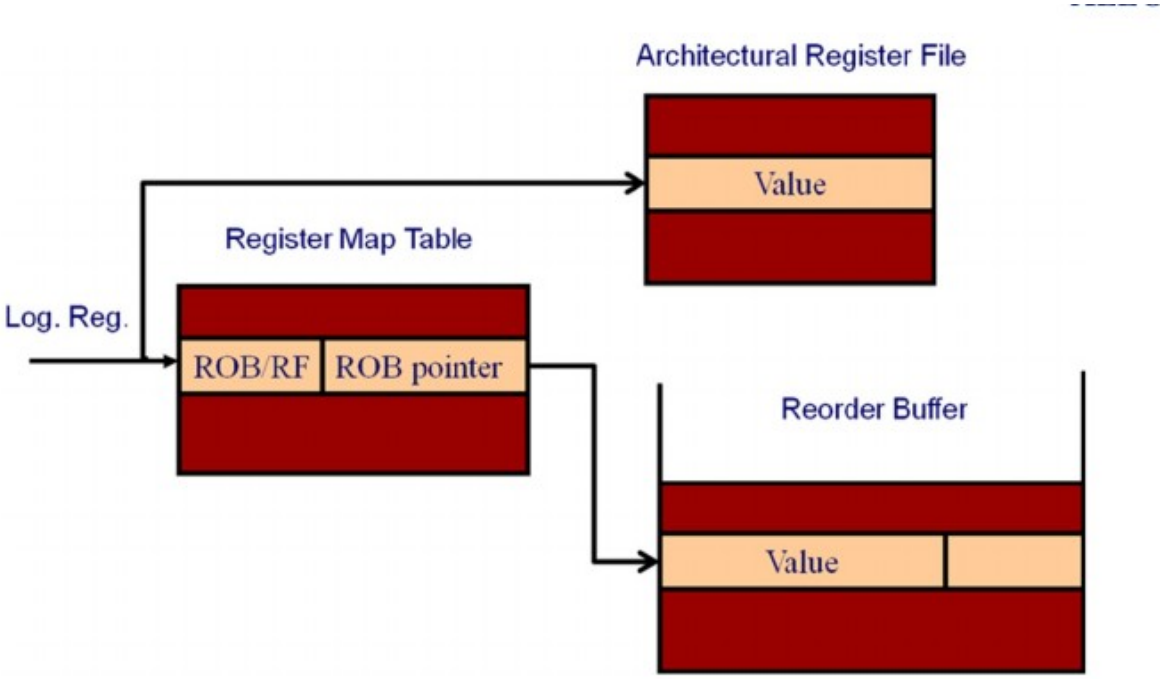


Figure 15: Rename through the reorder buffer

To avoid write after write hazard it is only need that commit stage avoids committing an instruction who produced a result which has been overwritten by a subsequent instruction so updating properly the ROB/RF bit in register map table.

This model was used by Intel Core 2 among others.

3.1.4 Renaming through a rename buffer

This scheme is a small variation of the previous one. In ROB we had an slot entry for each possible in-flight (noncommitted) instruction. But an important percentage of instructions do not produce any register result, it depends on the ISA but almost one third of instructions do not produce any result, so one third of ROB is wasted space.

To avoid this a rename buffer was introduced, which is a table like the ROB structure where entries are assigned to instructions who are producing values. When an instruction produces a value and there is no space left on rename buffer instruction is stalled. There's any chance of deadlocks since oldest instructions on pipeline cannot depend on newest

ones, so eventually an entry will be freed and instruction will proceed.

This scheme is used by IBM Power3 among others.

3.1.5 Merged register file

In this scheme, register file stores both speculative and committed values. Because of that number of registers is bigger than architectural registers.

Each register can be freed or allocated. When an instruction produces a result it is assigned a free physical register. If there's any free register then instruction is stalled until some older instruction frees one. To manage free registers we keep track of them in a separate list which can be implemented in several ways, for example with a circular list.
Each time a register is allocated we keep track on a rename table his associate logical register. The scheme can be shown in the following figure:

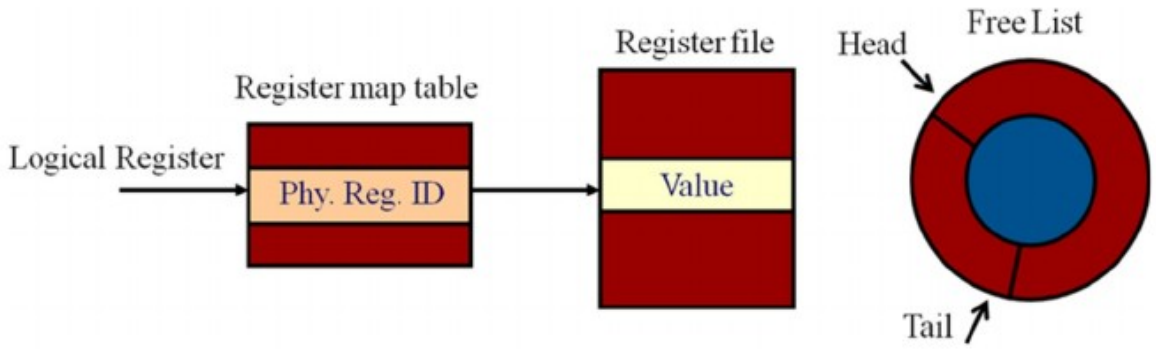
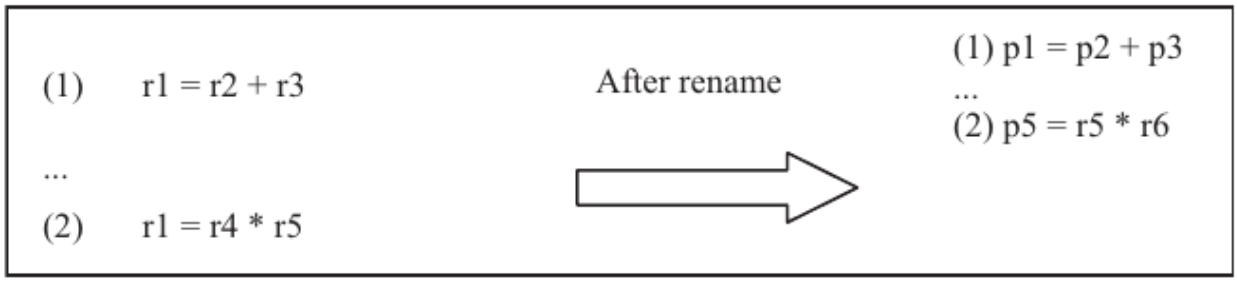


Figure 16: Merged register file

With rename table we search his source registers on there to find out which physical register was assigned and rename them properly.

Physical registers are freed when no further instruction will read them. This can be easily known by compilers but no by ISAs and so no for hardware. Because of that a conservative safe rule is used: a register is dead when the next instruction which uses the same architectural destination register commits. This can be explained with the following code:



Code 13: Releasing a physical register

Instruction 2 is overwriting the architectural destination register r1, so when it commits the assigned physical register associated with first instruction (p1 in this case) can be freed. Notice that it is not possible to free p1 when instruction 2 is fetched because it could be squashed due a branch misprediction, for example.

3.2 Proposal rename algorithm

As said previously we are going to try vectorizing repetitive instructions in loops. To vectorize an instruction there are some vectorial physical registers, usually of 128 bits (four 32 bit data slots), which needs to be filled in order to execute vectorial instructions.

Our design vectorizes scalar instructions that are repeated four times if they are not interdependent. Initially instructions are renamed with scalar registers, but vectorial units feed with vectorial registers, so when we get a vectorial instruction stride it is needed to rename registers properly into vectorial registers.

If we first rename in a conventional manner, and when we get our vectorial stride rename them again to fill up the vectorial registers slots, we are wasting scalar registers, and also wasting energy due to double renaming instructions. We guess that in a loop a lot of instructions can be vectorized so if we proceed this way we are for sure wasting energy misachieving our goal.

Due to this we propose to initially rename in a vectorial way, so assuming that instructions are part of a vectorial stride. But since we can be wrong, we need to be able accessing data in a scalar way. To do so we design a register file filled up only with vectorial registers, but with the particularity that we can access each of the vectorial slots individually as an scalar register.

This leads to a register file design where common registers are placed in four-column rows, which are the 128 bits of vectorial registers, and each matrix position is a typical scalar register. This scheme is shown in the next figure:

R0.0	R0.1	R0.2	R0.3
R1.0	R1.1	R1.2	R1.3
R2.0	R2.1	R2.2	R2.3
R3.0	R3.1	R3.2	R3.3

Figure 17: Register file schema

The point is that we can access each row as one single register and feed our vectorial unit with them, but we can also access each element of rows as single standard registers and feed the scalar units with them. The main advantage is that it will allow us to rename registers in a vectorial way without worrying if that instruction could not be vectorial, because if it isn't our rename will still be valid and no re-renaming will be needed.

To detect four vectorizable instructions the full window of instructions at each cycle is reviewed. We call window as the number of instructions passing through stages at the same time, which in other words is viewing instructions of each thread passing through the stage at a given cycle (referencing to superscalar architecture described in chapter 2.2.5). Also, since we want to see four repetitions of instructions, our proposed window is of 256

instructions (a large window) which statistically in most cases is enough to find that number of repetitions.

Proposed rename algorithm is based on the merged register file, and it only tries to detect vectorial arithmetic instruction strides with two source registers and one destination register explicitly declared, so instructions like branches and comparisons are treated as regular scalar instructions and so do all other instructions.

On our algorithm, for each instruction we look at his source registers producers. If we can see the producer (if it's on the same window) then we rename the producer destination register so each time it appears on rename stage it gets filling a row of register file until it is full. The following pseudocode attachment clarifies the algorithm explanation:

```
for each producer instruction do
 if not renamed(instruction) then
  if not assigned_row(instruction) then
   if(free_rows()) then
    row ←assign_row(instruction)
    rows_assigned(instruction) ←row
   else
     regular rename algorithm applies
   endif
  else
    current_row ← rows_assigned(instruction)
    current_position ←busy position(current_row)
    if(current_position = 3) then
       print “new vectorial word completed!”*****
       remove_rows_assigned(instruction)
       free_row(current_row)
    else
       add_busy_position(current_row)
     endif
  endif
 endif
endfor
```

Code 14: Proposed rename algorithm

Each time an instruction arrives, for each of its source-register producers, algorithm search for an unused row and assign its producer destination register first position of the row. The row gets marked as “trying to fill up a vectorial word” so regular scalar instructions will not get registers into these rows avoiding interferences. When instruction is again seen, since it has been marked in a separate structure with his assigned row, we know how many positions it has already fulfilled and can assign the next one.

When we get to the final fourth position, since the rename applies when some instruction uses it on our source registers, and we only apply the rename in arithmetic instructions with two source registers and one destination register, it means that four occurrences of instruction have the two source registers filled up in rows and in an aligned way, so we can

execute all four instances like a vectorial instruction (if there aren't any memory dependencies, which will be explained later in dispatch algorithm).

3.2.1 Rename algorithm execution example

Given the following code (illustrative purposes, it is not a real code):

```
i1. ldr r0, r4(r3)
i2. adds r2, r2, #1;
i3. ldr r1, r6(r3)
i4. adds r1, r0, r1
i5. str r1, r7(r3)
i6. adds r3, r3, #4
i7. b
i1. ldr r0, r4(r3)
i2. adds r2, r2, #1;
i3. ldr r1, r6(r3)
i4. adds r1, r0, r1
i5. str r1, r7(r3)
i6. adds r3, r3, #4
i7. b
[it repeats for 4 iterations]
```

Code 15: Rename algorithm sample code

As our window is of 256 instructions, all of three iterations are visible from rename stage.
The first instruction is a load, since it is not arithmetic we apply the regular rename to it.
Second instruction is an add, since r2 source operand we don't know which instruction produced it (it is not on our window), we apply regular rename on it.
Third instruction is a load, once again we apply regular rename.
For the fourth instruction we know producers of each of it's source registers. Register 0 is produced by a load which has not been applied ever on our rename algorithm, so we allocate for it a new row and rename i1 r0 destination register as R0.0. The same rule applies for the r1 source register, so i3 destination register gets allocated to R1.0. Code is as follows:

```
i1. ldr r0.0, r4(r3)
i2. adds r2, r2, #1
i3. ldr r1.0, r6(r3)
i4. adds r1, r0.0, r1.0
```

Code 16: Renamed first iteration

In this first iteration there aren't any other instructions fulfilling our rename conditions, so we get to the second one.
Now for the i2 instruction, since it has one immediate source value and it is not a register, it not fulfills our conditions so no rename algorithm gets applied to it.
We follow until again i4 is seen. The producer instructions are the same, since producers were already seen by our algorithm, we annotated that the last time first column were assigned, so we assign them the second column of their respective rows. The second

iteration gets renamed as follows:

```
i1_2. ldr r0.1, r4(r3)
i2_2. adds r2, r2, #1
i3_2 ldr r1.1, r6(r3)
i4_2. adds r1, r0.1, r1.1
```

Code 17: Renamed second iteration

The algorithm proceeds until we get to fourth iteration and we fill the last row position for each of i1 and i3. So the fourth iteration is as follows:

```
i1_4. ldr r0.3, r4(r3)
i2_4. adds r2, r2, #1
i3_4 ldr r1.3, r6(r3)
i4_4. adds r1, r0.3, r1.3
```

Code 18: Renaming completes vectorial word

We annotated we successfully complete a vectorial word – under rename point of view – and we can assign destination register of i4 a whole column, so all four i4 instructions could be executed like a vectorial instruction.

On register file the assigned registers are depicted in the following figure:

i1_1	i1_2	i1_3	i1_4
i3_1	i3_2	i3_3	i3_4
i4_1	i4_2	i4_3	i4_4

Figure 18: Register assignment per instruction

The other no renamed instructions get assigned register file rows which are marked as “scalar rows”, that is, it's positions are only used by regular instructions.

3.2.2 Correctness of vectorial words

It is important to notice that if on our process would be possible that one instruction to be vectorized had it's source registers unaligned, that is, in different columns, that would avoid us to vectorize them and our algorithm won't work.

This can not happen on our algorithm because of the following three reasons:

1. We only apply the algorithm on arithmetic instructions with two source registers and one destination register explicitly declared (so discarding branches, comparisons, etcetera).

2. Since we are searching occurrences of the same instance, it means that it have the same PC. So his source registers are referencing exactly to the same PC producer instruction each time. If still the PC producer were assigned to other row position between two occurrences, it would mean that in the same sequence of instructions

another instruction had happened renaming PC destination register in-between, which would mean breaking the PC sequence of instructions and that is not possible because program code do not change during execution.

3. Also notice that an instruction is renamed only one time, not two, so still in the case in which in the sequence of instructions were two instructions consuming the same destination register, only one of the instructions will be introducing our rename algorithm, the other one will find the producer instruction already renamed.

The fact that this can not happen implies that arithmetic instructions can be vectorized (if no memory dependencies occur, which will be explained later).

3.2.3 Additional structures of rename algorithm

So on our merged register file rename algorithm, the free register list gets a little more complex. It not only has to manage the free register, but keep track of rows which have not assigned instruction to it filling the row. To do this we keep the four bits on each row indicating which positions are free, if all positions are free, then the row is free.
Also we need a bit to indicate if row it's been using by regular scalar instructions – which are not tried to be vectorized – or not.

So register file structure is the following (the last row is used by instructions which are not going to be vectorized).

				Free bits	Scalar row
R0.0	R0.1	R0.2	R0.3	0000	0
R1.0	R1.1	R1.2	R1.3	0000	0
R2.0	R2.1	R2.2	R2.3	0000	0
R3.0	R3.1	R3.2	R3.3	0000	1

Figure 19: Register file design

We also keep track of instructions currently renamed with our algorithm and those who are not. So another structure is needed. To identify instructions we use it's PC (Program Counter) which is unique for each instruction.
Related to instructions we also need to know which row have assigned in the case they are renamed with our algorithm, so two informations can be merged into one single structure, depicted this way:

PC	Row assigned

Figure 20: PC row assignment table

To know if a PC has been renamed or not with our algorithm, it Is enough to search them into this structure, and at the same time we get it's assigned row.

To know which position have to be assigned into a PC already seen and renamed, it is enough to check the free bits of register file row. If the only free bit is the last one, we know we completed a word.

3.3 Dispatch stage

At this point the rename stage is actually renaming the registers in a vectorial way. But we are not really executing vectorial instructions.
Also be noticed that our rename it is not taking into account memory dependencies, for example assume the following loop iteration:

```
for(int c = 0; c<10; ++c) {
    A[i]  = A[i-1] + B[i];
}
```

Code 19: Non-vectorial code

And the generated assembly code:

```
i1. ldr.w r1, [r4, #4]!
i2. addi_uop.w r4, r4, #4
i3. adds r3, r3, #1
i4. ldr.w r2, [r6, #4]!
i5. addi_uop.w r6, r6, #4
i6. cmps r3, r5
i7. add r2, r2, r1
i8. str r2, [r4, #4]
i9. b
```

Code 20: Non-vectorial assembly code

Instruction 7 in every iteration will know which are it's source-registers producers. But actually it's r1 source operand is calculated by previous iteration, so instruction it's not really vectorizable. Rename algorithm do not manage this because it does not know about memory dependencies.

The stage who knows about memory dependencies and decides which instructions can be issued and which not is the dispatch stage, which fills the issue queue, telling issue stage which instructions can be executed.

Dispatch stage checks which source registers of instructions are ready (have his operands calculated) and which not. When an instruction have his operands ready can be issued, so instructions are dispatched.

Whenever an instruction finishes its execution and commits, dispatch stage updates the state of instructions waiting to be dispatched. It checks which instructions were waiting for the produced operand and mark these source registers as ready, so in the next cycle more instructions can be dispatched – those who have all his operands ready -.

3.4 Proposal dispatch modifications

To generate vectorial instructions we modify the dispatch stage. Since rename algorithm ensures that on dispatch all registers are placed on a vectorial way, our mission is to check

if vectorial words generate by rename can really be executed in a vectorial way – that is, there are no dependencies between same instance of instructions -. If this is the case, then a vectorial instruction is generated which will generate results for the scalar instances. So scalar instances will bypass issue and execute stages in parallel with vectorial instruction, without doing any execution. Once in commit stage vectorial instruction will disappear and scalar instructions will proceed committing normally.

It is the rename stage which tell dispatch which instructions are to be stalled. Those are the instructions which source operands are in the window of rename, that is the arithmetic operations that starts our rename algorithm.

So our dispatch proposal modification constis of one bit setted by rename stage which tells dispatch which instructions must be stalled. In pseudocode this modification could be the following function:

```
function stallInstruction(instruction)
  if(stalled_PC(instruction.PC)) then
    if(times_seen(instruction.PC) == 3) then
     add_to_block(instruction.PC,instruction)
     complete_block_wait_sources(instruction.PC)
     remove_times_seen_PC(instruction.PC)
   else
     add_to_block(instruction.PC,instruction)
     increment_times_seen(instruction.PC)
   endif
 else
   new_block_instructions(instruction.PC, instruction)
   increment_times_seen(instruction.PC)
 endif
endfunction
```

Code 21: Dispatch stall instructions algorithm

On the other hand we modify the general management of instructions in dispatch in the following way:

```
function dispatch
 for each instruction do
  if stall(instrtuction) then
    block instruction
  else
    regular dispatch algorithm
  endif
 endfor
endfunction
```

Code 22: Modified dispatch algorithm

And the last part which manages unstalling instructions:

```
function awakeInstructions
  for each block do
   ready_instructions = 0
   for each block instruction do
     if source_registers_ready(instruction) then
       ready_instructions ← ready_instructions + 1
    else
       abort
    endif
  endfor
  if(ready_instructions = 4) then
   create_vectorial_instructions(ready_instructions)
   unstall_block_instructions(ready_instructions)
  endif
 endfor
 for each stalled instruction do
   if(cycles_stalled(instruction) = 256) then
     release_whole_block(instruction.PC)
   endif
 endfor
endfunction
```

Code 23: Unstall dispatch instructions algorithm

There are two relevant parts of this code:

1. When we found a complete block of 4 instructions with all of source operands ready, we generate a vectorial instruction.

2. When an instruction reaches 256 cycles of starvation without being able to find his vectorial instruction, it is automatically unstalled. The release_whole_block() function just finds out all stalled instructions of the same PC and unstall them.

3.4.1 Additional structures of dispatch stage

As it happens with rename algorithm, for our dispatch modifications we need also some additional structures.

First, we must keep track of those instructions seen and how many times they have seen. Altogether with keeping track of their block (group of instruction instances). This structure picture is:

PC	Count	Block ID

Figure 21: Instruction instances counter

To keep track of blocks instances we have an structure like a register file, as depicted:

Block ID	Instruction 1	Ready?
Block ID	Instruction 2	Ready?
Block ID	Instruction 3	Ready?
Block ID	Instruction 4	Ready?

Figure 22: Blocks structure

Ready bit serves as reference about if the instruction instance has his source operands ready or not, so if it could be issued or not.

3.5 Second algorithm

For now on we have designed a microarchitecture based on keeping track of how many times an instruction instance is seen, grouping them and launching an equivalent vectorial instruction of them all.

But there has been developed a second algorithm very similar to this one. Instead of keeping track of how many times an instruction instance is seen, we keep track of how many times the same kind of instruction is seen. That is, if we have seen four additions, or four subtractions, etcetera, we can group them in a vectorial version of that instruction kind, it doesn't matter if they are the same instance or not.

The only change in the design is to replace PC for operation kind. Therefore the proposal design structures and algorithms are the same but replacing PC for instruction kind.

The main difference is that while in the first version we always need a loop to find the same PC, in the second one we don't need any loop, just several instructions of the same kind close enough. So our goal is to evaluate if programs use in larger percentages the same kinds of instructions in a way that it could be vectorized or not, since if that were the case, then with this alternative algorithm we expect to achieve better results.

4. Implementation

The standard procedure designing microprocessors is to first implement it in a computer architecture simulator. This is because physical designs are expensive, so the cost of an inefficient physical design is huge and not affordable, while implement a design in a simulator is cheaper, since it's costs are hardware needed to run he simulator – some computers - and the human resources expended.

So our implementation will be through a simulator. Since there are several computer architectures simulator it is needed to choose one.

One of these simulators is PTLSim. It is a widely known simulator of x86 architectures, and it has several real pipeline implementations like the ones for Intel Core 2 or AMD K8. It supports syscall emulation mode, which is a kind of emulation where only the user space is simulated and simulator provides directly system call services, and full system emulation where codes can be executed inside a running emulated kernel which could be a Linux one, for example, so user and system space are simulated, with no need for simulator to provide system calls.

In PTLSim, full system emulation mode is achieved through a Xen hypervisor, which is a platform to run virtual machines.

Other simulator is gem5, which has been developed recently and it is growing its community support. It supports running x86 architectures, as well as an ARM and ALPHA.

Gem5 is a more complete simulator since it also supports several memory management systems, and supports implementing peripherals interfaces like could be a monitor or a printer.

Also it has several CPU models implemented, not only x86 architectures but also simple non-pipelined architectures, multi-threaded out-of-order architectures and some many others.

It also supports syscall emulation mode and full system emulation mode, in this case full system is not done through virtualization which achieves better performance and accurate statistic data.

Main drawback about gem5 it's that since it is a very recent work it lacks a lot of official documentation and explanation, it only has like 30% of the simulator explained. On the other hand it has a growing community which can offer support through open mailing lists - which are open and accessible for everyone – so it can compensate the absence of official documentation on some parts.

Since gem5 is more complete and flexible in modeling several computer architecture components, which can be helpful in future work on the design and also can allow modeling a real full computer architecture for evaluation, we choose to implement our design in gem5 despite the fact of missing official documentation and guides.

4.1 Gem5 simulator architecture

Gem5 is mainly written in C++ and Python.
C++ is used to code the several components internals, like the CPU models, memory management systems, and devices.
Python on the other hand is used to configure the parameters of these components, like how many physical registers the CPU has or the memory size. Python is also used to describe the architecture on which some simulation will be ran, defining its CPU model, memory and connected peripherals.

The figure below shows the function call sequence for the initialization of gem5 when running the "Hello, World" example (included in gem5 source code). Each node in the graph gives the path to the source code file in which the function is located and helps to understand how the configuration files are used and how the objects are constructed:

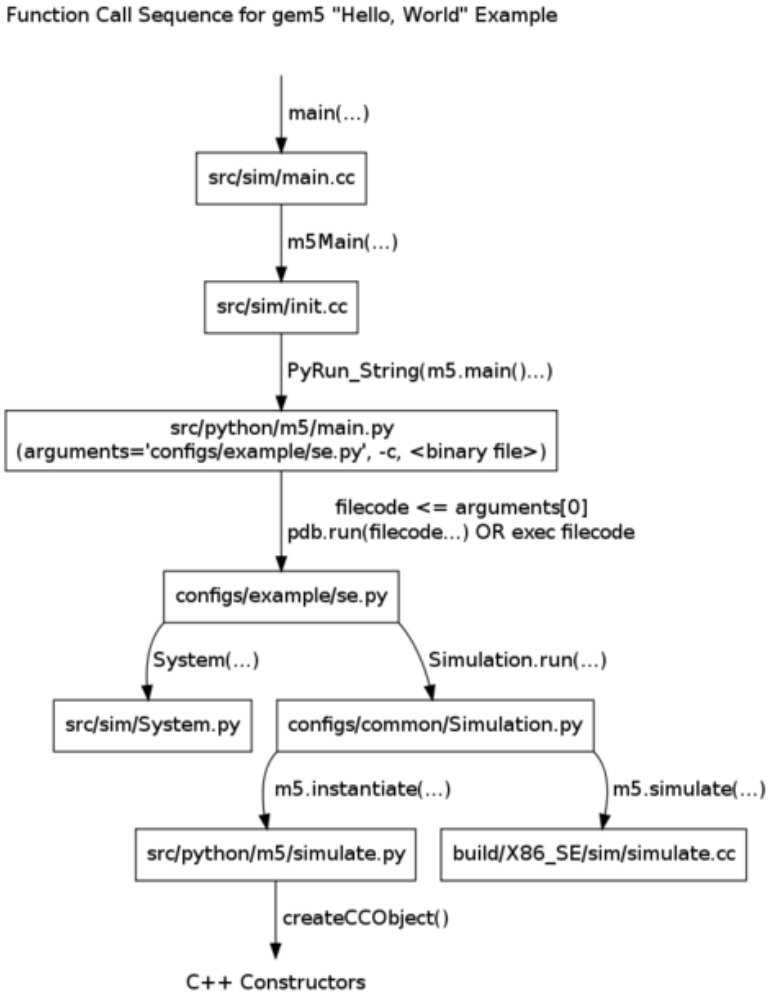


Figure 23: gem5 Initialization Function Call Sequence

It can be appreciate that first a C++ main code is called. This code initializes the M5 simulator part (remember, gem5 is a mix between M5 and GEMS simulators). Then it calls the configuration python script where the computer architecture to be used is set. To do that C++ code includes a python interpret inside, so all python scripts are interpreted inside a C++ basic simulator core.
Within the configuration python script, computer architecture is defined, like the CPU model, the memory management system to be used, etcetera. After being defined computer architecture configuration script calls the instantiate function which starts a code mapping python simulation objects to the C++ objects – since components are written with that language -, which invokes C++ constructors. On the other hand, system function invokes basic fundamental simulator functionalities such as events handlers, statistics utilities, syscalls emulators – needed when ran in syscall emulation mode -, etc.
Once script defined everything it call simulate function which starts simulation itself.

There are also several other languages on subcomponents not depicted here, since they are part of bigger C++ components. For instance, since this simulator allows defining several ISAs, it has it's own language (C based language) to define the ISA instructions.

4.1.1 Simulator source code tree

To get an idea about how simulator is structured we look at source-code tree:

- **AUTHORS** – A list of people who have historically contributed to gem5.
- **LICENSE** - The license terms that apply to gem5 as a whole, unless overridden by a more specific license.
- **README** - Some very basic information introducing gem5 and explaining how to get started.
- **SConstruct** - A part of the build system, as is the build_opts directory.
- **build_opts** - holds files that define default settings for build different build configurations. These include X86_FS and MIPS_SE, for instance.
- **configs** - Simulation configuration scripts which are written in python. The files in this directory help make writing configurations easier by providing some basic prepackaged functionality. Here is where our simulated computer architecture is defined.
- **ext** - Things gem5 depends on but which aren't actually part of gem5. Specifically, dependencies that are harder to find, not likely to be available, or where a particular version is needed. For example, here we can find a code implementing the UDP network protocol.
- **src** - gem5 source code.
    - **arch** - ISA implementations.
        - **generic** - Common files for use in other ISAs.
        - **isa_parser.py** - Parser that interprets ISA descriptions.
        - **ISA directories** - The files associated with the given ISA.
            - **OS directories** - Code for supporting an ISA/OS combination, generally in SE mode.
            - **isa** - ISA description files.
    - **base** - General data structures and facilities that could be useful for another project.

        - **loader** - Code for loading binaries and reading symbol tables.
        - **stats** - Code for keeping statistics and writing the data to a file or a database.
        - **vnc** - VNC support.
    - **cpu** - CPU models.
    - **dev** - Device models.
        - **ISA directories** - Device models specific to the given ISA
    - **doxygen** - Doxygen is a C++ source-code documentation tool, and here there are it's templates & output.
    - **kern** - Operating system specific but architecture independent code (e.g. types of data structures).
        - **OS directories** - Code specific to the given simulated operating system.
    - **mem** - Memory system models and infrastructure.
        - **cache** - Code that implements a cache model in the classic memory system.
        - **ruby** - Code that implements the ruby memory model.
        - **protocol** - Ruby protocol definitions.
        - **slicc** - The slicc compiler.
    - **python** - Python code for configuration and higher level functions.
    - **sim** - Code that implements basic, fundamental simulator functionality.
- **system** - Low level software like firmware or bootloaders for use in simulated systems.
    - **alpha** - Alpha console and palcode.
    - **arm** - A simple ARM bootloader.
- **tests** - Files related to gem5's regression tests.
    - **configs** - General configurations used for the tests.
    - **test-progs** - "Hello world" binaries for each ISA, other binaries are downloaded separately.
    - **quick, long** - Quick and long regression inputs, reference outputs, and test specific configuration files, arranged per test.
- **util** - Utility scripts, programs and useful files which are not part of the gem5 binary but are generally useful when working on gem5.

4.1.2 Running simulations with gem5

To run a simulator we must define our system architecture. This is done via a python scripts located at configs directory.

The scripts need to define the simulator components to be used and specific parameters if needed – otherwise the default ones will be used – and they are in charge to trigger simulator execution of specified code.

For instance, one of the basic examples is a script which defines a CPU model of 2Hz with ruby memory management system with 512 MB of memory and no devices at all, running on syscall emulation mode. This script allows setting some parameters on execution time as defining which CPU model to use as well as defining customized parameters such as if

there is instruction and data cache or not – and it's sizes -, some checkpointing code utilities among others. This is a very useful tool of configuration scripts since it may allow running several architectures simulations to provide better design evaluation.

Another interesting examples are those which defines some network topoliges like a P2P network or a taurus one, showing the ability of the simulator to simulate things beyond basic computer architectures.

We also must compile a version of gem5. To do this is necessary to build a simulator target among the five provided which are the followings:

- **gem5.debug:** has optimizations turned off. This ensures that variables won't be optimized out, functions won't be unexpectedly inlined, and control flow will not behave in surprising ways. That makes this version easier to work with in tools like gdb, but without optimizations this version is significantly slower than the others. It it must be used only for debugging purposes.
- **gem5.opt:** has optimizations turned on and debugging functionality like asserts and DPRINTFs left in. This gives a good balance between the speed of the simulation and insight into what's happening in case something goes wrong. This version is best in most circumstances.
- **gem5.fast:** has optimizations turned on and debugging functionality compiled out. This pulls out all the stops performance wise, but does so at the expense of run time error checking and the ability to turn on debug output. This version is recommended if you're very confident everything is working correctly and want to get peak performance from the simulator.
- **gem5.prof:** is similar to gem5.fast but also includes instrumentation that allows it to be used with the gprof profiling tool. This version is not needed very often, but can be used to identify the areas of gem5 that should be focused on to improve performance.
- **gem5.perf:** also includes instrumentation, but does so using google perftools, allowing it to be profiled with google-pprof. This profiling version is complementary to gem5.prof, and can probably replace it for all Linux-based systems.

To build our target we also must select a set of compile-time build options that control simulator functionality, such as the ISA used, which CPU models are included, which Ruby coherence protocol to use, etc. Of course, this must be done in coherence with configuration script selected. We cannot run an O3CPU model in a build target which only compiles the simple CPU model.

Every time we made modifications in simulator code components, will be needed to rebuild our target, although only the components modified not all components.

With build target compiled and configuration script, we have all needed to run our programs.

4.1.3 CPU models

Our proposal design is modifying a proccessor design on it's rename and dispatch stages, so before implementing it we have to decide which CPU model to use, or if we need to build a new one.

Gem5 gives us the following CPU models already implemented:

- **Simple CPU**: this model is just a linear processor, with no segmentation at all. So it is the most basic processor kind as described in chapter 2.2.2.

- **Inorder CPU**: this model is a scalar in-order processor as described in chapter 2.2.3. It is based on O3CPU.

- **O3CPU**: this model is a superscalar out-of-order processor as described in chapter 2.2.5, and so the closer to nowadays processor design. Also it has more implementation documentation than the others. It is highly parameterizable via configuration script, it allows changing basic things like physical architecture registers and complex things like commit stage policy.

Since O3CPU is the most complete one and closer to real processors, we choose to implement our design on that model.

4.1.4 O3CPU model

The model pipeline is depicted this way:

Fetch	Decode	Rename	Dispatch	Issue Execute and Writeback	Commit	Retire

Figure 24: Stages of O3CPU

In this model implementation, issue, execute and writeback stages are treated as a single stage. Also it has an extra stage which is retire one, in charge of removing instructions from ROB once they are completed and some other instruction producing same logical register commits, ensuring that no subsequents data hazards will need to check that value.

Stages work as explained on chapter 2.4.

The following picture depicts the most important classes of O3CPU pipeline and relations between them:

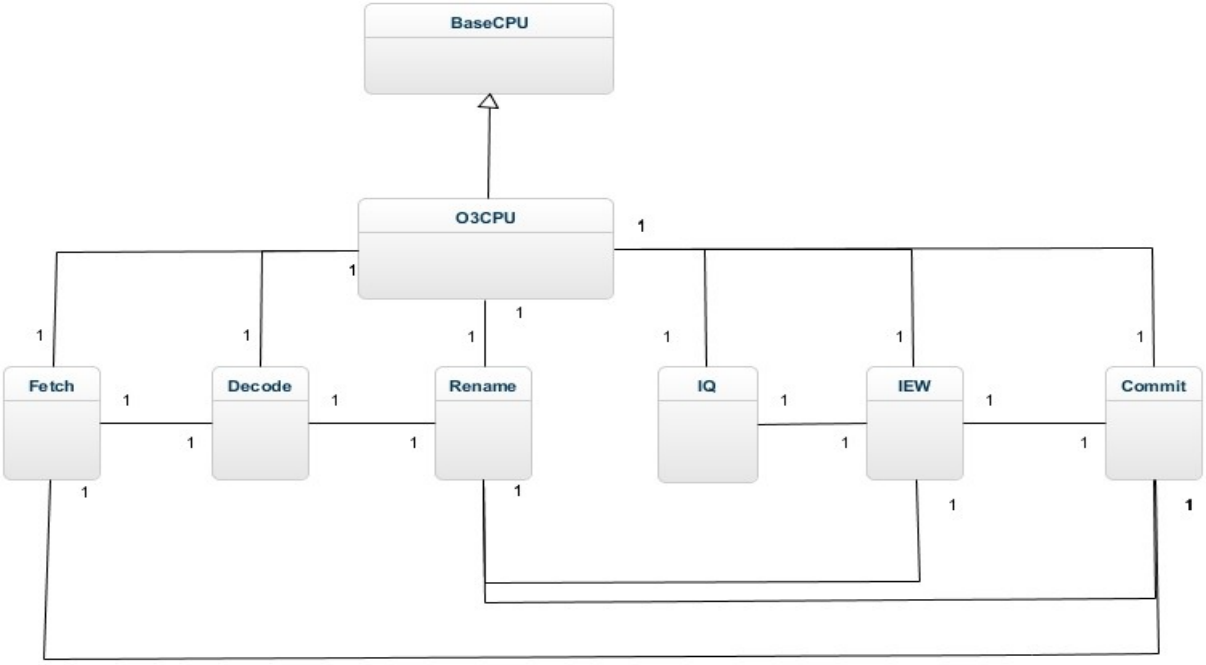


Figure 25: O3CPU Class diagram

There are several other classes, which are structures needed for each of the stages, such as ROB or load/store queue. We do not depict them in the diagram since they are part of basic classes, and we do not need modifying them on our implementation.

CPU class is mandatory for CPU models on gem5. Each CPU model needs to have a CPU class which derives from BaseCPU. This class is used to communicate with other architecture components, such as memory or devices. So when fetch stage decides that some address needs to be load from instructions cache, it asks for it to CPU class which has the connection ports with memory system.

Dispatch stage is executed at “iew” (issue, execution and writeback) stage. It is implemented on instruction queue class (IQ in diagram). Instruction queue also manages dependencies between instructions and is in charge of issuing instructions who has their source registers ready, which are the functions of dispatch stage.
Notice that rename do not has a direct relationship with instruction queue, instead it has with “iew”, this is due because it is the “iew” who calls IQ.

4.1.4.1 O3CPU Rename algorithm

The O3CPU rename algorithm works, briefly and on pseudocode in order to simplify, as follows:

```
function rename
  foreach instruction in decode to rename bus do
     if renamed instructions lesser rename bus bandwidth then
        rename destination registers
        rename source registers
       renamed instructions ← renamed instructions + 1
     else
        do nothing
     endif
  endfor
endfunction

function rename destination registers
 foreach destination register do
   if free registers on free registers queue then
     rename registers
   else
     stall instruction
     mark instruction as waiting free registers
    endif
 endfor
endfunction

function rename source registers
 foreach source register do
    renamed register number ← lookup source register on rename
map
   source register ← renamed register number
   if scoreboard_ready(source register) then
      mark source register as ready
   endif
 endfor
endfunction
```

Code 24: O3CPU rename algorithm

This pseudocode only focuses on relevant parts for our implementation. Rename algorithm also checks if there are any stalled instructions in previous cycles and tries to unstall them. Also checks if there was some branch misprediction or memory mispeculation in order to fix rename history and retire mispredcited instructions, etcetera. But any of these details are not relevant to our proposal design, since we only change how we are renaming registers, and so we do not explain them.

4.1.4.2 O3CPU Dispatch algorithm

Dispatch algorithm is in charge of manage dependencies between instructions and to keep track of instructions with all his source registers ready – with data available –. Once an instruction has his source operands ready it is placed on the ready to issue queue, and it'll be issued whenever issue stage has free slots and earlier ready to issue instructions get issued.

Basic functions pseudocode – in a simplified way – are the following:

```
function wakeDependents(instruction)
  foreach destination register of instruction do
     dependants ← dependents list (destination register)
      foreach dependents do
         mark source register as ready
          addIfReady(dependent instruction)
      endfor
  endfor
endfunction

function addIfReady(instruction)
  if instruction source registers are ready then
     add to ready to issue queue
  endif
endfunction

function scheduleReadyInsts()
  foreach ready to issue instruction do
     if issued instructions < to issue bandwidth then
        issue instruction
     endif
  endfor
endfunction
```

Code 25: O3CPU dispatch algorithm pseudocode

4.2 Rename implementation

Our proposal design is based on a rename algorithm, and a dispatch algorithm. So first we start implementing the rename algorithm.

4.2.1 Register file structure

Our rename algorithm needs a new register file structure, since our designed register file is structured in a matrix way. We also need to ensure that rename stage is able to read at least four full iterations of, at least, medium sized loops, so rename bus bandwidth needs to be changed.

A first attempt was made implementing register file changes on simulator pipeline, but it was seen that it was trickier than it seemed at a first glance. That structure is used by CPU not only to provide data to IEW, but also it provides data to memory system when there's a

load or an store. Suddenly, what it was assumed to be a 10 hours development turned out in a 30 hours unsuccessful development with crashes on simulator.
Since we only need statistical data, providing how many words rename stage is able to complete, it was no needed to actually change register file. So new additional structures were build around current pipeline to simulate another register file, only used to calculate how many vectorial words rename is able to complete. At this point, instead of increasing window size of rename stage, to achieve this effect there was also implement a parallel structure who keep instructions information for 256 cycles, simulating rename window.

Our implementation register file has the following structure representing rows:

```
struct RenameInfo {
     int numReg; //Position within row
     int fila; //Row assigned
     int ciclesTotals; //Cycles waiting
   };
```

Code 26: Register file implemented structure

That is the basic information for each renamed register, it says on which row it has been asigned (fila), which position within row (numReg), and also we store information about how many cycles an instruction has been waiting to has it's row complete (ciclesTotals).

And to keep track the rows assigned per PC, we have the following C++ map:

```
std::map<long long int, RenameInfo> paraules
```

Code 27: Mapping between PC and register file rows

This structure, alongside with “cicles” information in RenameInfo, helps us to simulate our window of instructions. Since every producer PC has a row associated, and within that row we know how many cycles we have been keeping it, it helps us determine if there has been too much cycles waiting to complete a word for a producer PC. Therefore, when a number of cycles – 256 in our initial implementation – have executed keeping that PC information, it is considered that rename do not longer is able to rename instances destination registers, so to apply our algorithm.
This is equivalent to say that our rename is able to see 256*bus_width instructions previous to the current one, since assuming each cycle we are able to view a full bus width of instructions, in 256 cycles we have seen 256*bus_width instructions. If instead we have a real bus width of 256*bus_width, we are renaming in a single cycle the same number of instructions.

On the other hand we maintain a list of free rows to keep track of which rows are free and which are not:

```
std::list<int> files;
```

Code 28: Free rows list

This list is initially of 1024 rows, so we are assuming a register file of 4096 registers (4 registers per row). Whenever a row is fully completed, it is mark as a vectorial word completed and row is freed, going back to the front of list.

4.2.2 Instructions information structure

We also need to keep associating registers with its producing instructions, to know which instruction is producing our source values, which triggers renaming algorithm as explained in proposal design 3.2.

```
std::map<PhysRegIndex, DynInstPtr> dependencies;
```

Code 29: Dependencies tracking

As rename algorithm also stalls instructions on dispatch stage, it maintains an structure to keep track of how many cycles instructions are delayed:

```
struct delayedInfo {
        int cicles;
        long long int pc;
        DynInstPtr instr;
    };
```

Code 30: Delayed instructions information

4.2.3 Rename algorithm

Now we have all structures needed to implement our proposal design. It only last algorithm implementation itself. At the following code we rename source registers in a vectorial way, this is our core implementation:

```
template <class Impl>
inline void
DefaultRename<Impl>::renameSrcRegs(DynInstPtr &inst, ThreadID tid)
{
    assert(renameMap[tid] != 0);

    bool delayed = false;
    // Get the architectual register numbers from the source and
    // destination operands, and redirect them to the right register.
    // Will need to mark dependencies though.
    for (int src_idx = 0; src_idx < num_src_regs; src_idx++) {
        RegIndex src_reg = inst->srcRegIdx(src_idx);
        RegIndex flat_src_reg = src_reg;

        if (src_reg < TheISA::FP_Base_DepTag) {
            flat_src_reg = inst->tcBase()->flattenIntIndex(src_reg);
            DPRINTF(Rename, "Flattening index %d to %d.\n",
                    (int)src_reg, (int)flat_src_reg);
        } else if (src_reg < TheISA::Ctrl_Base_DepTag) {
            src_reg = src_reg - TheISA::FP_Base_DepTag;
            flat_src_reg = inst->tcBase()->flattenFloatIndex(src_reg);
          DPRINTF(Rename, "Flattening index %d to %d.\n",
```

```
                    (int)src_reg, (int)flat_src_reg);
            flat_src_reg += TheISA::NumIntRegs;
        } else if (src_reg < TheISA::Max_DepTag) {
            flat_src_reg = src_reg - TheISA::Ctrl_Base_DepTag +
                           TheISA::NumFloatRegs + TheISA::NumIntRegs;
            DPRINTF(Rename, "Adjusting reg index from %d to %d.\n",
                    src_reg, flat_src_reg);
        } else {
            panic("Reg index is out of bound: %d.", src_reg);
        }

        // Look up the source registers to get the phys. register they've
        // been renamed to, and set the sources to those registers.
        PhysRegIndex renamed_reg = renameMap[tid]->lookup(flat_src_reg);

      //If it is not load nor store, and nor branch or cmps and explicit
register
        if(tipus!=MemReadOp && tipus!=MemWriteOp && inst->staticInst->getName()!
="b"
         && inst->staticInst->getName()!="cmps" && src_reg != TheISA::ZeroReg)
      {
          //If we know which is it's producer instruction
         if(dependencies.find(renamed_reg)!=dependencies.end()) {
            long long int direccio = dependencies[renamed_reg]->instAddr();
/** DEBUG PURPOSES
            if(inst->seqNum>=14153 && inst->seqNum<14320) {
               sim_meva = true;
            } else
               sim_meva = false;
*/
            //Its producer must not be itself, otherwise crash
            assert(dependencies[renamed_reg]->seqNum != inst->seqNum);
            /*** HERE STARTS OUR RENAME ALGORITHM CORE **/
            if(!inst->staticInst->isMicroop() /*&& sim_meva*/) {
              //Stall instruction if not already stalled
              if(!delayed) {
                    delayed = true;
                    long long int dirInst = inst->instAddr();
                    //Tell dispatch to stall instruction
                    iew_ptr->instQueue.delayInstruction(inst);
                    //iew_ptr->instQueue.delayedInsts[inst->seqNum] = true;
                    delayedCicles[inst->seqNum].cicles = 0;
                    delayedCicles[inst->seqNum].pc = dirInst;
                    delayedCicles[inst->seqNum].instr = inst;
                      assert(iew_ptr->instQueue.delayedInsts.find(inst->seqNum)!
=iew_ptr->instQueue.delayedInsts.end());
                    ++numDelayedInsts;
                  }
              //Check if PC producer already saw and renamed
              if(paraules.find(direccio)==paraules.end()) {
               //If not assign a new row
                RenameInfo rename;
                rename.numReg = 1;
                rename.cicles = 0;
                rename.ciclesTotals = 0;
                assert(!files.empty());
                rename.fila = files.front();
                files.pop_front();
                paraules[direccio] = rename;
```

```
              } else {
                //Seen and this instruction completes a word
                if(paraules[direccio].numReg==3) {
                 // std::cerr<<paraules[direccio].cicles<<",";
                  paraules[direccio].numReg = 0;
                  paraules[direccio].cicles = 0;
                  //Statistic information, new word completed
                  ++renameVectInstsC4;
                  //Free row
                  files.push_front(paraules[direccio].fila);
                } else {
                  //Next row position used
                  paraules[direccio].numReg++;
                }
              }
            }
            //Destination register already renamed, so we erase it from our
information
              dependencies.erase(dependencies.find(renamed_reg));
          }
      }
      /** FINISH RENAME ALGORITHM CORE */

      DPRINTF(Rename, "[tid:%u]: Looking up arch reg %i, got "
                "physical reg %i.\n", tid, (int)flat_src_reg,
                (int)renamed_reg);

        inst->renameSrcReg(src_idx, renamed_reg);

        // See if the register is ready or not.
        if (scoreboard->getReg(renamed_reg) == true) {
            DPRINTF(Rename, "[tid:%u]: Register %d is ready.\n",
                    tid, renamed_reg);

            inst->markSrcRegReady(src_idx);
        } else {
            DPRINTF(Rename, "[tid:%u]: Register %d is not ready.\n",
                    tid, renamed_reg);
        }

        ++renameRenameLookups;
        inst->isFloating() ? fpRenameLookups++ : intRenameLookups++;
    }
}
```

Code 31: Rename source registers algorithm

Notice a condition where we check source registers are not equal to TheISA::ZeroReg, this is a simulator reference to implicit registers. Implicit registers are those like zero conditions flags, or other special register information which are stored within instruction information for simplicity puroposes on CPU model implementation.
We only want to rename explicit source registers, which are the “real” source registers of instructions, so all implicit registers associated are not triggering our rename algorithm.

Here is the rename destination registers algorithm. Our modification only consists on maintaining the dependencies structure (code 28) used at previous code.

```
template <class Impl>
inline void
DefaultRename<Impl>::renameDestRegs(DynInstPtr &inst, ThreadID tid)
{
    typename RenameMap::RenameInfo rename_result;

    unsigned num_dest_regs = inst->numDestRegs();

    // Rename the destination registers.
    for (int dest_idx = 0; dest_idx < num_dest_regs; dest_idx++) {
        RegIndex dest_reg = inst->destRegIdx(dest_idx);
        RegIndex flat_dest_reg = dest_reg;
        if (dest_reg < TheISA::FP_Base_DepTag) {
            // Integer registers are flattened.
            flat_dest_reg = inst->tcBase()->flattenIntIndex(dest_reg);
            DPRINTF(Rename, "Flattening index %d to %d.\n",
                    (int)dest_reg, (int)flat_dest_reg);
        } else if (dest_reg < TheISA::Ctrl_Base_DepTag) {
            dest_reg = dest_reg - TheISA::FP_Base_DepTag;
            flat_dest_reg = inst->tcBase()->flattenFloatIndex(dest_reg);
            DPRINTF(Rename, "Flattening index %d to %d.\n",
                    (int)dest_reg, (int)flat_dest_reg);
            flat_dest_reg += TheISA::NumIntRegs;
        } else if (dest_reg < TheISA::Max_DepTag) {
            // Floating point and Miscellaneous registers need their indexes
               // adjusted to account for the expanded number of flattened int
regs.
            flat_dest_reg = dest_reg - TheISA::Ctrl_Base_DepTag +
                            TheISA::NumIntRegs + TheISA::NumFloatRegs;
            DPRINTF(Rename, "Adjusting reg index from %d to %d.\n",
                    dest_reg, flat_dest_reg);
        } else {
            panic("Reg index is out of bound: %d.", dest_reg);
        }

        inst->flattenDestReg(dest_idx, flat_dest_reg);

        // Get the physical register that the destination will be
        // renamed to.
        rename_result = renameMap[tid]->rename(flat_dest_reg);

        //MODIFICATION CODE HERE
        if(dest_reg != TheISA::ZeroReg)
        {
           dependencies[rename_result.first] = inst;
        }
        //END MODIFICATION

        //Mark Scoreboard entry as not ready
        if (dest_reg < TheISA::Ctrl_Base_DepTag)
            scoreboard->unsetReg(rename_result.first);

        DPRINTF(Rename, "[tid:%u]: Renaming arch reg %i to physical "
                "reg %i.\n", tid, (int)flat_dest_reg,
                (int)rename_result.first);

        // Record the rename information so that a history can be kept.
        RenameHistory hb_entry(inst->seqNum, flat_dest_reg,
                               rename_result.first,
```

```
                                rename_result.second);

        historyBuffer[tid].push_front(hb_entry);

        DPRINTF(Rename, "[tid:%u]: Adding instruction to history buffer "
                "(size=%i), [sn:%lli].\n",tid,
                historyBuffer[tid].size(),
                (*historyBuffer[tid].begin()).instSeqNum);

        // Tell the instruction to rename the appropriate destination
        // register (dest_idx) to the new physical register
        // (rename_result.first), and record the previous physical
        // register that the same logical register was renamed to
        // (rename_result.second).
        inst->renameDestReg(dest_idx,
                            rename_result.first,
                            rename_result.second);

        ++renameRenamedOperands;
    }
}
```

Code 32: Rename destination registers algorithm

Finally, there is one modification left. At the end of each cycle we have to do 2 new things:

1. Update information about out-date producer instructions. That is, those instructions that have been trying to complete a vectorial word 256 unsuccessfully needs to be removed.

2. Those instructions that have been stalled on dispatch stage 256 needs to be freed (that is, to renounce completing a vectorial word and issue them). This is done on rename stage for simplicity purposes, and it does not have any relevance which stage keeps track of how many cycles instructions have been stalled.

```
   //(1) For each instruction on register file
   for(Itparaules it = paraules.begin(); it!=paraules.end(); ++it)
    {
        //If it has been on our window for 256 cycles, update statistic
information about vectorial words and remove it
        it->second.ciclesTotals++;
        if(++it->second.cicles==256) {
          if(it->second.numReg == 1)
            //Uncompleted one single slot filled
           ++renameVectInsts;
          else if(it->second.numReg == 2)
           ++renameVectInstsC2; //Uncompleted two slots filled
          else
           ++renameVectInstsC3; //Uncompleted three slots filled

          //DPRINTF(ElmeuFlag, "Netejo PC: 0x%08llx \n", it->first);
         //Remove instruction and update free rows list
          files.push_front(it->second.fila);
          paraules.erase(it);
        }
    }
//(2) For each delayed instruction
    for(Itcicles it = delayedCicles.begin(); it!=delayedCicles.end(); ++it)
```

```
    {
       //If it has been delayed for 56 cycles, unstall it
        if(it->second.cicles == 56)
        {
        //if still it's stalled on dispatch, unstall
                            if(iew_ptr->instQueue.delayedInsts.find(it->first)!
=iew_ptr->instQueue.delayedInsts.end()) {
             iew_ptr->instQueue.awakeDelayedInst(it->second.instr);
           }
         //Remove from record of delayed instructions
           delayedCicles.erase(it->first);
        } else {
           //Check if, maybe, it is part of a stalled block of four instructions
ready to issue. This is done on dispatch stage
                            if(iew_ptr->instQueue.delayedInsts.find(it->first)!
=iew_ptr->instQueue.delayedInsts.end())
             iew_ptr->instQueue.awakeIfBlock(it->second.instr);
          //Update cycles stalled information
           it->second.cicles++;
      }
    }
```

Code 33: Rename updates on each cycle

4.3 Dispatch implementation

Now we have rename algorithm implemented, we can implement dispatch algorithm.

4.3.1 Additional structures

Firs thing we need is to know how many times we saw a stalled instruction PC. Since when we have saw four times the same PC, a new vectorial instruction may be done.

To do this, we code the following structures:

```
struct delayInfo {
   std::list<DynInstPtr> instructions;
   int times;
 };
std::map<long long int, delayInfo> timesDelayed;
```

Code 34: PC instances counter

For each PC, we store the associated instruction instances saw (instructions) and how many times the same PC have been found (times).

Once we have found four instances of the same PC, we also need to group then in some structure, what we call a block of instances, which will be waiting to be ready at the same time. This is because we cannot launch a new vectorial word with the instances until all of them are ready to issue in their scalar versions, which will mean that their source registers are ready.

So, we also generate some simple structure to manage those blocks:

```
std::map<InstSeqNum, std::list<DynInstPtr> > blockDelayed;
```

Code 35: Block management structure

Notice that we are mapping a sequence number of instruction. Each instruction that enters pipeline is assigned a unique sequence number identifying it. That is what we use to map this structure. The last instruction completing a block is used as the sequence number, and the list of instructions contains all the instruction pointers that were stored in delayInfo structure.

Last, we need to keep track of those instructions we are stalling in order to avoid issuing them earlier.

```
std::map<InstSeqNum, int> delayedInsts;
```

Code 36: Delayed instructions structure

We only need the unique identifier – sequence number – of the instructions, and we map with number of cycles we have been stalling it.

4.3.2 Dispatch algorithm implementation

To stall instructions is quite simple: in original addIfReady function, which is called every time simulator tries to push an instruction onto ready instructions queue, we only must ensure that if the coming instruction is on the list of delayed ones, do not allow to be pushed onto queue, so it will not be issued.

The modification can be seen in the following code:

```
template <class Impl>
void
InstructionQueue<Impl>::addIfReady(DynInstPtr &inst)
{
    /** ADDED CODE */
    //We search if instruction must be stalled. If so set delayed to true
    bool delayed = false;
    if(delayedInsts.find(inst->seqNum)!=delayedInsts.end())
    {
      delayed = true;
    }
    /** END ADDED CODE */
     //We add delayed to false condition to really insert to ready instructions
queue
    // If the instruction now has all of its source registers
    // available, then add it to the list of ready instructions.
    if (!delayed && inst->readyToIssue()) {

        //Add the instruction to the proper ready list.
        if (inst->isMemRef()) {

            DPRINTF(IQ, "Checking if memory instruction can issue.\n");

            // Message to the mem dependence unit that this instruction has
            // its registers ready.
            memDepUnit[inst->threadNumber].regsReady(inst);

            return;
        }
```

```
        OpClass op_class = inst->opClass();

        DPRINTF(IQ, "Instruction is ready to issue, putting it onto "
                "the ready list, PC %s opclass:%i [sn:%lli].\n",
                inst->pcState(), op_class, inst->seqNum);

        readyInsts[op_class].push(inst);

        // Will need to reorder the list if either a queue is not on the list,
        // or it has an older instruction than last time.
        if (!queueOnList[op_class]) {
            addToOrderList(op_class);
        } else if (readyInsts[op_class].top()->seqNum  <
                   (*readyIt[op_class]).oldestInst) {
            listOrder.erase(readyIt[op_class]);
            addToOrderList(op_class);
        }
    }
}
```

Code 37: Dispatch addIfReady function modifications

Now we have to manage the following:

1. Manage instructions needed to be stalled. Asked from rename stage.

2. Manage those instructions that must be awaked. Once an instruction must be awaken, we have to check if it is part of a group of blocked instructions, and if it's the case awake all instructions to ensure no register misalignment can occur.

3. Check if some instruction has it's source operands ready, and if it's the case and it was the last instruction from a block, awake all block since a new vectorial instruction could be generated.
   Notice that it is not possible that other block instructions do not have their source operands ready as they producers are the same instructions due to the basic correctness principle explained on chapter 3.2.2.

4. Increment the cycles counter that has been stalled. This is achieved through a very simple function.

5. If a load is blocked an one of its source operands its an stalled one, unstall it.

All of this is managed through 4 new functions and the last one adding some extra code on the dependencies manager of the dispatch stage.

1. Function to stall instructions.

```
template <class Impl>
void
InstructionQueue<Impl>::delayInstruction(DynInstPtr inst)
{
    //Add to delayedInsts
    delayedInsts[inst->seqNum] = 0;
    //If PC already seen, PC seen counter + 1
```

```
    if(timesDelayed.find(inst->instAddr())!=timesDelayed.end())
    {
      //If PC seen 4 times, build new block
      if(++timesDelayed[inst->instAddr()].times == 4) {
         timesDelayed[inst->instAddr()].instructions.push_back(inst);
      blockDelayed[inst->seqNum] = timesDelayed[inst->instAddr()].instructions;
         //Erase old structure
         timesDelayed.erase(inst->instAddr());
         assert(blockDelayed[inst->seqNum].size()>0);
      } else {
        timesDelayed[inst->instAddr()].instructions.push_back(inst);
      }
    } else {
      timesDelayed[inst->instAddr()].times = 1;
      timesDelayed[inst->instAddr()].instructions = std::list<DynInstPtr>();
      timesDelayed[inst->instAddr()].instructions.push_back(inst);
   }
   assert(delayedInsts.find(inst->seqNum)!=delayedInsts.end());
}
```

Code 38: Delay instructions management

2. Function to manage awakening of instructions.

```
template <class Impl>
void
InstructionQueue<Impl>::awakeDelayedInst(DynInstPtr inst)
{
   //If it is part of a block, release all block
  assert(delayedInsts.find(inst->seqNum)!=delayedInsts.end());
    if(blockDelayed.find(inst->seqNum)!=blockDelayed.end())
    {
      for(ListIt lit = blockDelayed[inst->seqNum].begin();
            lit != blockDelayed[inst->seqNum].end(); ++lit)
      {
        DynInstPtr awake = *lit;
        if(delayedInsts.find(awake->seqNum)!=delayedInsts.end()) {
          delayedInsts.erase(awake->seqNum);
          addIfReady(awake);
        }
      }
      blockDelayed.erase(inst->seqNum);
    } else { //else only release itself
       delayedInsts.erase(inst->seqNum);
       addIfReady(inst);
    }
};
```

Code 39: Awake delayed instructions management

3. Function to awake ready blocks.

```
template <class Impl>
void
InstructionQueue<Impl>::awakeIfBlock(DynInstPtr inst)
{
 //If ready to issue and last instruction of block, release block
 if(inst->readyToIssue() && blockDelayed.find(inst->seqNum)!=blockDelayed.end())
```

```
    {
       awakeDelayedInst(inst);
    }
};
```

Code 40: Awake instruction if block ready to issue

4. Function which increments cycles counter of stalled instructions.

```
template <class Impl>
void
InstructionQueue<Impl>::addDelayedCycles()
{
    for(DelayedIt it = delayedInsts.begin(); it != delayedInsts.end(); ++it)
      it->second++;
}
```

Code 41: Increment cycles stalled

5. Modification of addToDependents dispatch function, to unstall instructions producing source operands of load instructions.

```
template <class Impl>
bool
InstructionQueue<Impl>::addToDependents(DynInstPtr &new_inst)
{
    // Loop through the instruction's source registers, adding
    // them to the dependency list if they are not ready.
    int8_t total_src_regs = new_inst->numSrcRegs();
    bool return_val = false;

    for (int src_reg_idx = 0;
         src_reg_idx < total_src_regs;
         src_reg_idx++)
    {
        // Only add it to the dependency graph if it's not ready.
        if (!new_inst->isReadySrcRegIdx(src_reg_idx)) {
            PhysRegIndex src_reg = new_inst->renamedSrcRegIdx(src_reg_idx);

            // Check the IQ's scoreboard to make sure the register
            // hasn't become ready while the instruction was in flight
            // between stages.  Only if it really isn't ready should
            // it be added to the dependency graph.
            if (src_reg >= numPhysRegs) {
                continue;
            } else if (regScoreboard[src_reg] == false) {
                DPRINTF(IQ, "Instruction PC %s has src reg %i that "
                        "is being added to the dependency chain.\n",
                        new_inst->pcState(), src_reg);

            /** MODIFICATION: if it is a load, if it source operand is on our
stalled instructions, unstall it
            if(new_inst->opClass() == MemReadOp)
            {
                 if(delayedInsts.find(dependGraph.getProducer(src_reg)->seqNum)!
=delayedInsts.end())
                {
                   DPRINTF(ElmeuFlag,  "Trobo  load  amb  productor  bloquejat.
Desbloquejar              productor              PC:              0x%llx\n",
```

```
dependGraph.getProducer(src_reg)->instAddr());
                  awakeDelayedInst(dependGraph.getProducer(src_reg));
               }
            }
            /** END MODIFICATION */
                dependGraph.insert(src_reg, new_inst);

                // Change the return value to indicate that something
                // was added to the dependency graph.
                return_val = true;
            } else {
                DPRINTF(IQ, "Instruction PC %s has src reg %i that "
                        "became ready before it reached the IQ.\n",
                        new_inst->pcState(), src_reg);
                // Mark a register ready within the instruction.
                new_inst->markSrcRegReady(src_reg_idx);
            }
        }
    }

    return return_val;
}
```

Code 42: addToDependants dispatch modification

Be noticed that we are not issuing any vectorial instruction, nor we are reconverting any instruction to hat. Since we were only interested on statistical data, we only collect information about how many cycles instructions waited to find four same instances of instructions renamed in a vectorial way and ready to issue, and when we have done that we get that information, which is the relevant statistical information. Implementing a real reconversion to a vectorial instruction implied a lots of more implementation work which won't have been useful since the relevant statistics will have been the same ones.

4.4 Second algorithm implementation

We are not going to show the code of second algorithm since it is almost the same as for the first one. The only difference is replacing all the PC-based operations to a name-based operations, where the name is the instruction kind (i.e, “addition” or “subtraction”). Therefore it's codification is straightforward.

To show a sample of how this is coded, the mapping between PC and register file row did in rename, is replaced by the following code:

```
std::map<string, RenameInfo> paraules
```

Code 43: Mapping between instruction kind and register file rows

We only replaced “long long int” (the C++ type of PC) per “string” (the C++ type for names).
And a sample of the rename source registers where the vectorial words are builded:

```
//If we didn't sind any instruction of this kind, use a new row
                  if(paraules.find(name)==paraules.end()) {
                    RenameInfo rename;
                    rename.numReg = 1;
                    rename.cicles = 0;
                    rename.ciclesTotals = 0;
                    assert(!files.empty());
                    rename.fila = files.front();
                    files.pop_front();
                    paraules[name] = rename;
                  } else {
//Else, if we complete a row, update statistical data
                    if(paraules[name].numReg==3) {
                      paraules[name].numReg = 0;
                      paraules[name].cicles = 0;
                      ++renameVectInstsC4;
                    } else {
                      paraules[name].numReg++;
                    }
                  }
```

Code 44: Second algorithm core rename algorithm sample

As it can be seen, the modifications are only replacing PC by instruction kinds. The rest of the algorithm is exactly the same.

5. Validation and evaluation

In the next lines we are gonna run three benchmarks to validate the design implementation and evaluate the potential of our technique.

5.1 Code semantics validation

To validate our implementation do not break code semantics we used a simple code where thousands of additions, multiplications and other operations are done with the following pregenerated numbers:

```
/* “Random” numbers initialization */
for (i=0;i<n;i++) {
 A[i]=i^(i<<6)^i;
 B[i]=(i-5)^(i<<4)^(i+3);
}
```

Code 45: Validation random generation

Where A and B are the vectors which positions are the operands of arithmetic instructions, and results stored in a third vector. This is shown in the following code:

```
int i,j;
for (j=0; j<N_iter; j++)
  for (i=0;i<len;i++)
   C[i] = A[i] + B[i];
```

Code 46: Validation algorithm

Then, we print the results vector (C vector) in a regular computer and we match results with those calculated by our simulator. As our results matched those of the simulator we can say our implementation do not change code semantics.

This validation check were executed before doing any modification, to ensure initial simulator results were correct – they were – and regularly at each simulator relevant code modification change.

5.2 Evaluation framework

Our evaluation gem5 configuration script will be simulating the following architecture:

1. O3CPU with 2GHz frequency and 4096 physical integer registers.
2. Simulated rename window of 256 instructions.
3. Stalling instructions on dispatch up to 56 cycles.
4. Gem5 ruby memory system and 512 MB memory, which includes both instructions and data cache.
5. ARM architecture of 64 bits.

5.3 Benchmark A – Vectorial loop

Given the following loop:

```
for(int c = 0; c<10; ++c) {
    A[i] = B[i] + C[i];
}
```

Code 47: Vectorial loop benchmark

And the produced ARM assembly code:

```
ldr r0, r4(r3) ;r0 = B[i]
adds r2, r2, #1 ;r2 = i
ldr r1, r6(r3) ;r1 = C[i]
adds r1, r0, r1
str r1, r7(r3) ;r7 = @A
adds r3, r3, #4 ;r3 = i+4
cmps r2, r5 ;r5 = 10
b
```

Code 48: Vectorial loop assembly clode

Easily can be seen that this loop is fully vectorial, since additions on each iteration do not depend on results of the others. So we could ran all the additions at the same time and the results would be correct.

5.3.1 Rename algorithm validation

To validate our results are correct, we ran our simulator limiting rename and issue changes only to our area of interest (the loop itself). So in the rest of the code we will not rename instructions in a vectorial way nor stall instructions on purpose.

First, we count how many vectorial words rename stage can achieve. The results are:
20 vectorial words and 4 rows which were trying to complete a vectorial word but only one column was filled up.

To verify this, we specify some debug flags on our simulator code and force it to give us which registers is renaming to, and also we get the simulated assembly code pipeline. This way we can know if some instruction was seen on rename window or not, and if the renamed registers are correct. Samples generated indicate for associated instruction which register file is assigned to its destination register.

The full samples are on appendix A.1.1, but here we give a few lines to analyze:

```
7762500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.0
7762500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.0
7762500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.0
7762500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.0
7763000: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.1
7763000: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.1
7763000: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.1
7763000: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.1
```

```
7763500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.2
7763500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.2
7763500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.2
7763500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.2
7764000: system.cpu.rename: PC: 0x00008986   adds   r2, r2, #1 C4 completat, assigno R0.3
7764000: system.cpu.rename: PC: 0x00008984   ldr   r0, [r4, r3] C4 completat, assigno R1.3
7764000: system.cpu.rename: PC: 0x00008988   ldr   r1, [r6, r3] C4 completat, assigno R2.3
7764000: system.cpu.rename: PC: 0x0000898e   adds   r3, r3, #4 C4 completat, assigno R3.3
7764500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.0
7764500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.0
7764500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.0
7764500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.0
7765000: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.1
7765000: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.1
7765000: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.1
7765000: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.1
7765500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.2
7765500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.2
7765500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.2
7765500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.2
7766000: system.cpu.rename: PC: 0x00008986   adds   r2, r2, #1 C4 completat, assigno R0.3
7766000: system.cpu.rename: PC: 0x00008984   ldr   r0, [r4, r3] C4 completat, assigno R1.3
7766000: system.cpu.rename: PC: 0x00008988   ldr   r1, [r6, r3] C4 completat, assigno R2.3
7766000: system.cpu.rename: PC: 0x0000898e   adds   r3, r3, #4 C4 completat, assigno R3.3
```

Code 49: Renaming full eight iterations

Here we can see eight full iterations.
The first thing we appreciate is that we are renaming in a vectorial way two additions with immediate values. This were not on our proposal design, but as explained in design implementation, it was easier to implement this way, and since an immediate value could be filled in a registers row with it's values, or via some bypasses to ALU, it is possible to design it in a processor, and indeed it would be better this way.

On the other hand, we also can appreciate how the same addition is consuming produced value of previous iteration, triggering our rename on his registers. That instruction is not really vectorizable, but as explained rename can not know about that fact and so it renames guessing it could be vectorial.
Also with this instruction can be noticed that first iteration of loop cannot trigger anything, rename algorithm starts on second iteration where producers of source registers are on rename window.

The two loads which destination registers are renamed are due to adds r1, r0, r1. This is the real vectorial addition as explained, and it is triggering the proper rename on its source registers to the proper register file slots. Since we have its source register on the proper columns we could ran it as a vectorial word. So we almost validate our rename algorithm is working, we only need to calculate the full instruction execution and validate the rename words results.

A simple manual check of the sample gives us 8 vectorial words in 8 iterations, so each 4 iterations we have 4 completed vectorial words. That is correct. Checking the whole sample showed in appendix A.1.1 gives us the 20 vectorial words. To understand the extra iterations, it can be shown on our pipeline: it is due to iteration branch. Since branch is predicted to be taken on tenth iteration – it was on the last nine – we fail. This fail implies that we are getting through pipelines some incorrect instructions, and since our bandwidth bus is of 8 instructions – the exact size of an iteration - due to branch evaluation latency we get many full mistaken iterations getting inside rename stage, where it starts filling

some vectorial words. Exactly 25 iterations get into rename, so completing 12 vectorial words and leaving one uncompleted word with only one column filled. So the numbers match.

Our rename implementation works fine on this benchmark.

5.3.2 Dispatch algorithm validation

We also must check that we are stalling and unstalling instructions properly. That is, when a group of 4 instances of the same instruction is found to be stalled, it must be unstalled if they source registers are ready. And if none of the conditions apply to one instruction after 56 cycles – implementation parameter which can be changed to anyone else – it must be unstalled alongside with his other stalled instances – to avoid register misalignment -.

To validate it, first we need to look at assembly code and find real vectorizable instructions. There is only one: adds r1, r0, r1. It is the only one whose source registers changes every iteration and it's values do not depend on the previous ones.

Also, looking at the assembly code we expect to try vectorizations on the other to additions, but as they depends on the previous iterations calculus, they are expected to be blocked and automatically freed after 56 blocking cycles.

To validate all of this we proceed in the same way as for rename algorithm validation: generating a debug trace (full trace can be seen at Appendix A.1.2):

```
7762500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7762500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7762500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7763000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7763000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7763000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7763500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7763500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7763500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7764000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7764000: system.cpu.iq: New block of 4 instances completed
7764000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7764000: system.cpu.iq: New block of 4 instances completed
7764000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7764000: system.cpu.iq: New block of 4 instances completed
7764000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7764000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7764500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7764500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7764500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7764500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7764500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7765000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7765000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7765000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7765000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7765000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7765500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7765500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7765500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7765500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7765500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7766000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7766000: system.cpu.iq: New block of 4 instances completed
7766000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7766000: system.cpu.iq: New block of 4 instances completed
```

```
7766000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7766000: system.cpu.iq: New block of 4 instances completed
7766000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7766000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7766500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7766500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7766500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7766500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7766500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7767000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7767000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7767000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7767000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7767000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7767500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7767500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7767500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7767500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7767500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7767500: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 10 cicles
7767500: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 9 cicles
7767500: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 8 cicles
7767500: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 7 cicles
```

Code 50: The additions are stalled and those possible successfully vectorized

Here we can see that the blocked instructions are the three identified additions, and that only one of the additions is unblocked before 56 cycles time limit completes. So only the identified addition “add r1, r0, r1” can be launched as a vectorial word instruction.

Be noticed the delay between stalling instructions and freeing them. This is due because we are blocking “add r3, r3, #4”, which is producing data that will be used later by a load instruction, a necessary load to complete our vectorizable instructions. Due to this, in design implementation we release those stalled instructions which are producing results for loads and this can be seen on our debug trace. Only after some cycles of releasing this instruction, it completes and the four additions can have their source operands ready, and so identified as a vectorizable instruction.

The other addition which is not released because of producing a load, is released after the 56 cycles time limit – which can be seen on complete stack trace on appendix -.

Therefore our dispatch algorithm is working as expected.

5.3.3 Evaluation

Once our algorithms are validated, we need to evaluate how good they are. In order to do that we have to define some metrics.

For rename algorithm, our metric will be how many vectorial words can benchmarks complete, which will give us information about how vectorial is our code.

On the other hand we want to know how really vectorial is our code. That is, given the fact that rename stage completes words which could not be executed in a vectorial instruction due to memory dependencies, we have to know how many of them could be. This can be measured within dispatch stage, getting the count of how many instructions are unblocked before 56 cycles – our time limit – which are the instructions which are really vectorial.
Also at this point is interesting to know the average cycles instructions must be stalled before they are vectorized.

Also, as we want to compare global performance, the proper measure is the known CPI (cycles per instruction). We want our implementation CPI and the original CPI, of course, and compare them.

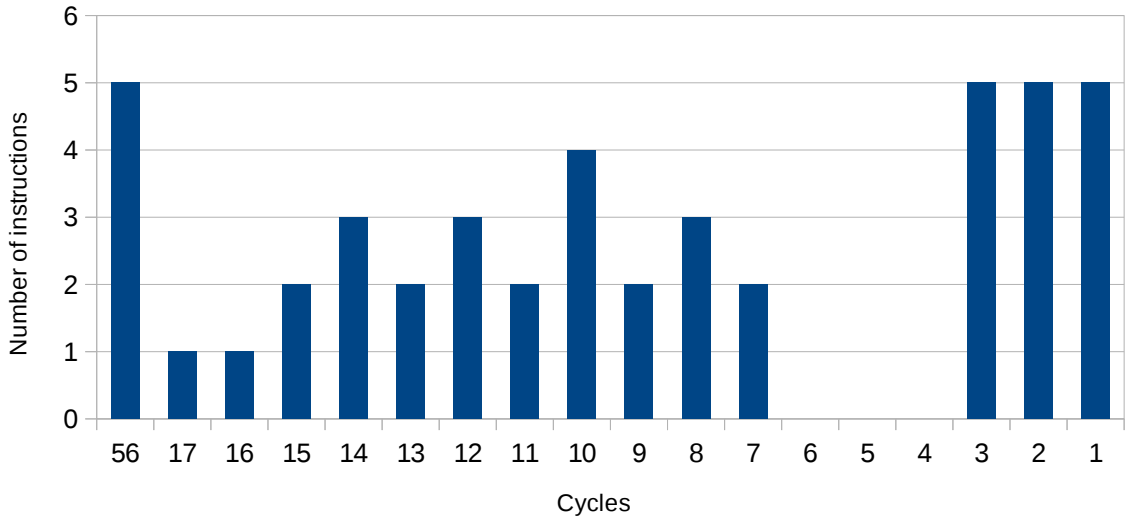
Results are as follows:

Rename completes 20 vectorial words, get 4 uncompleted words, and stalls 59 instructions.

Mean waiting cycles: 27 cycles.
Mean waiting cycles per instruction only of released on time instructions: 11 cycles.
27 instructions were stalled and released due to cycle limit.
CPI: 3.43

The next chart shows how much we wait to form a vectorial instruction since we find the first instruction of the vectorial one, that is the cycles we have to wait to find 4 instructions of the same PC ready to issue.

A[i] = B[i] + C[i]



The results were as expected, we produced a lot of words (in proportion to the only 10 iterations of the loop), and we find out that they were really vectorial. The oversize of words is explained because of the buses bandwidth configuration of the simulator, as explained in validation. The dispatch stage can found the “add r1, r0, r1” instruction as a vectorial one, also as expected and validated.

5.4 Benchmark B - Non-vectorial loop

Given the following loop:

```
for(int c = 0; c<10; ++c) {
    A[i]  = A[i-1] + B[i];
}
```

Code 51: Non-vectorial benchmark

And the produced ARM assembly code:

```
ldr.w  r1,  [r4,  #4]!  ;r4  =
A[i-1]
addi_uop.w r4, r4, #4
adds r3, r3, #1 ;r3 = i
ldr.w r2, [r6, #4]! ;r6 = B[i]
addi_uop.w r6, r6, #4
cmps r3, r5 ;r5 = 10
add r2, r2, r1 ;r2 = A[i-1] +
B[i]
str r2, [r4, #4] ;r4 = @A[i-1]
b
```

Code 52: Non-vectorial loop assembly code

It is clear that the loop is no vectorial since each iteration results depends on the produced by the previous one. However, since rename stage is not aware of that fact renaming algorithm should be able to allocate several vectorial words, and dispatch stage should not to be able to vectorize any of these words because of the dependencies between them, so stalling processor for long times.

5.4.1 Rename algorithm validation

As in previous benchmark using debug flags we check the renames being doing by simulator. There is a sample track trace (whole trace on Appendix A.2.1):

```
7679500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.0
7680500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.0
7680500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.0
7681500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.1
7682500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.1
7682500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.1
7683500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.2
7684500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.2
7684500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.2
7685500: system.cpu.rename: PC: 0x0000897e   adds   r3, r3, #1 C4 completat, assigno R0.3
7686500: system.cpu.rename: PC: 0x00008980   ldr.w   r2, [r6, #4]! C4 completat, assigno R1.3
7686500: system.cpu.rename: PC: 0x0000897a   ldr.w   r1, [r4, #4]! C4 completat, assigno R2.3
7687500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.0
7688500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.0
7688500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.0
7689500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.1
7690500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.1
7690500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.1
7691500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.2
7692500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.2
7692500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.2
7693500: system.cpu.rename: PC: 0x0000897e   adds   r3, r3, #1 C4 completat, assigno R0.3
```

```
7694500: system.cpu.rename: PC: 0x00008980   ldr.w   r2, [r6, #4]! C4 completat, assigno R1.3
7694500: system.cpu.rename: PC: 0x0000897a   ldr.w   r1, [r4, #4]! C4 completat, assigno R2.3
```

Code 53: Renaming four iterations successfully

Here the only weird thing that could be find is the order in which instructions are being renamed. Due to ldr.w r1, [r4, #4]! is being renamed after the other two candidate instructions to be renamed although it is the first one in the assembly code sequence. This is due to an implementation detail.

The implementation checks the source registers of instructions, and this is what triggers renaming destination registers of producers. Since in the code r1 is read after r3 and r2, the load destination register is renamed last. Since we are evaluating the potential of the technique we only want statistical data, and this implementation detail do not change that data, so it does not matter in which order we proceed on design implementation.

The total amount of renamed words is 10 fully completed vectorial words, and 3 uncompleted. This is again due because of the last branch mispredicitons and the bus bandwidth parameters as in the previous benchmark.

5.4.2 Dispatch algorithm validation

In this case the code is not vectorial, so any instruction will be released before 56 cycles. Due to this, our sample debug trace is the pice where instructions are freed. Completed strack trace can be found on Appendix A.2.3.

```
7707500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7707500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 0 cicles
7708500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7708500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 0 cicles
7709500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7709500: system.cpu.iq: New block of 4 instances completed
7709500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 0 cicles
7710500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7710500: system.cpu.iq: New block of 4 instances completed
7710500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 0 cicles
7711500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7711500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 0 cicles
7712500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 0 cicles
7713500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 12 cicles
7713500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 8 cicles
7713500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 4 cicles
7713500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 56 cicles
```

Code 54: Dispatch detects that the stalled iterations are not vectorial

Here it can be noticed that some instructions say that they are released before it's 56 cycles. This is due because they are part of a group of 4 instructions, and there is an instruction who has been there already 56 cycles (it can be seen the last one always says being released after 56 cycles). Due to this, all the group is released and so some instructions do not wait until 56 cycles, otherwise we will be getting misalignment problem as explained on dispatch design, section 3.4.

5.4.3 Evaluation

Rename completes 10 vectorial words, get 4 uncompleted words, and stalls 36 instructions.

Mean waiting cycles per instruction: no instruction could be released on time. No instruction were really vectorial at all.
36 instructions were stalled and released due to cycle limit.
CPI: 3.42

As this code were not vectorial, results are as expected as no stalled instruction can be vectorized.

5.5 Benchmark C – Non detectable vectorial loop

This benchmark code is as follows:

```
for(int c = 0; c<10; ++c) {
      A[i] = A[B[i]] + A[i];
}
```

Code 55: Non detectable vectorial loop benchmark

And the produced ARM assembly code:

```
ldr.w r0, [r6, #4]! ;r0 = @B[i]
addi_uop.w r6, r6, #4
adds r3, r3, #1 ;r3 = i
ldr r1, [r2, #0] ;r1 = A[i]
cmps r3, r5 ;r5 = 10
ldr.w r0, [r4, -r0 LSL #2] ; r0
= A[B[i]
add r1, r1, r0 ; r1 = A[i] +
A[B[i]]
str.w r1, [r2] #4 ;A[i] = A[i]
+ A[B[i]]
addi_uop.w r2, r2, #4
b
```

Code 56: Non-detectable vectorial loop assembly code

In this loop the B vector is initialized as follows: 0,n-1,1,n-3,...,n-2,1. In this case we are exploiting the fact that compilers cannot understand that this code is vectorial (because they cannot check memory positions referenced by pointers), but our technique does. So we expect a lot of vectorial words and that they are really vectorial, since looking at the referenced positions no iteration depends on previous ones.

5.5.1 Rename algorithm validation

As we are on a loop, we also expect to find out rename completes several vectorial words. We proceed in the same way as previous algorithms. Here is the sample trace (whole trace on Appendix A.3.1):

7869500: system.cpu.rename: Assigno a PC: 0x00008a28 adds r3, r3, #1 el registre R0.0

```
7869500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7869500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7870500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7870500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7870500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7871500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7871500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7871500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7872500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7872500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7872500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7873500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7873500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7873500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7874500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7874500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7874500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7875500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7875500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7875500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7876500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7876500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7876500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7877500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7877500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7877500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7884000: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7884000: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7884500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7884500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7885500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7885500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7885500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7886500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7886500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7886500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7887500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7887500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7887500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7888500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7888500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7888500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7889500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7889500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7889500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7890500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
```

Code 57: Renaming iterations successfully

It can be appreciated how register renaming is assigning registers properly in the order instructions arrive. The only weird thing is that suddenly some instructions do not appear on the pipeline and so they are no renamed. This is due to a branch misprediction on the first iteration, and because of the bus widths explained previously some instructions are not arriving to rename stage. Therefore, in the following correctly taken iterations we do not complete all vectorial words of the three candidate instructions at the same time.

5.5.2 Dispatch algorithm validation

In this case we expect be vectorizing instructions of “add r1, r0, r1”, since it is the only vectorial instruction. Also we expect to block the other addition, adds r3, r3, #1, but we can not be able to generate any vectorial instruction with that instruction.

Here is a sample of debug trace (full trace on Appendix A.3.2):

```
7869500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7869500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
```

```
7870500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7870500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7871500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7871500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7872500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7872500: system.cpu.iq: New block of 4 instances completed
7872500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7872500: system.cpu.iq: New block of 4 instances completed
7873500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7873500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7874500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7874500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7875500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7875500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7876500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7876500: system.cpu.iq: New block of 4 instances completed
7876500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7876500: system.cpu.iq: New block of 4 instances completed
7876500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 14 cicles
7876500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 12 cicles
7876500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 10 cicles
7876500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 8 cicles
7877500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
```

Code 58: Dispatch detects that the stalled iterations are vectorial although compiler coud not

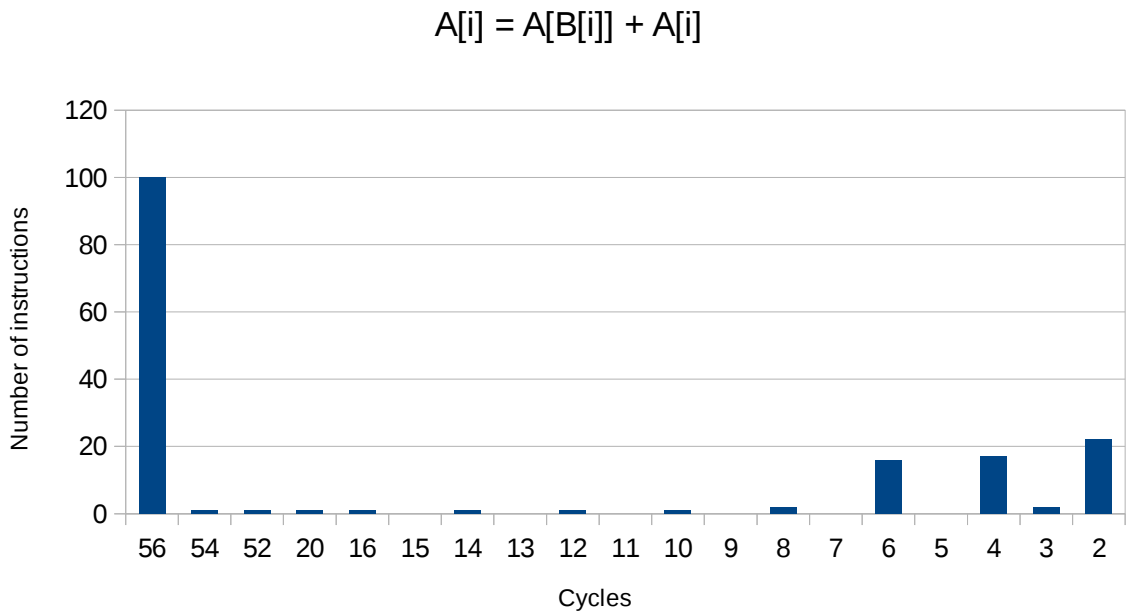
Easily we can see that our predicted results are fulfilled. Also in full stack trace on Appendix A.3.2 it can be shown how at the end these additions are freed after 56 cycles. Also some of the vectorial additions are freed, these are part of mispredicted iterations and are freed because his producer loads instructions do not commit and so his source registers never get ready to be vectorized.

5.5.3 Evaluation

Rename completes 32 vectorial words, and stalls 166 instructions.

Mean waiting cycles: 34 cycles.
Mean waiting cycles only of released on time instructions: 3 cycles.
100 instructions were stalled and released due to cycle limit.
CPI: 3.64

The chart of number of instructions per waiting cycles is as follows:



It finds out that the words are vectorial, something that a compiler could have not detected since it needs runtime data – which obviously cannot take -. This proves the point of our design: achieving some instructions vectorizations that a compiler could not achieve.

5.6 Performance evaluation

Finally we must evaluate global performance of processor. Since examples were only of small sized loop, and evaluating only those iterations, CPI results are not relevant, therefore comparing only that it is not enough.
Instead, to take a relevant sample of how our proposal design performs, we must enable all algorithms in the whole program – which includes calling C++ libraries code, for instance –. When we do this simulator stops because simulate limit cycles reached. This means that we are stalling so much instructions that even only stalling them for 56 cycles is not enough due to dependencies propagation, therefore consequent stallings takes so much cycles that simulator stops simulating. A sample of CPI at several millions of simulated cycles shows a CPI around 400 cycles.

This is happening because we stall instructions indiscriminately, trying vectorizing instructions without any idea if it is really possible. Therefore, when we stall a lots of instructions which are not vectorizable, they also stall instructions waiting for the operands. This is a domino effect and carries on stalling longer than the 56 cycles limit on the whole pipeline, and CPI gets increased in an exponential way.

This is totally unacceptable and proves that this algorithm doesn't performs good enough, although being capable of vectorize instructions in a dynamical way.

We discuss further about evaluation in conclusions (chapter 8).

5.7 Second algorithm

To validate second algorithm we proceeded the same way as before for rename algorithm, and we saw that, although algorithm worked as expected, vectorial words were misaligned. This is due because the correctness key of our algorithm do not apply in this case. The key is that if we are PC-vectorizing instructions, the producer of a PC is always the same instruction, and it was explained in chapter 3.2.2 that this fact guarantee that no misalignment in words can be produced. However, in this second version we are not grouping by PC but by kind of instruction, so the producers of two grouped instructions can be different, and almost always they will.

On the other hand our processor also executes memory instructions out-of-order, which means that sometimes loads execute before all previous stores addresses have been calculated. Only at commit stage processor is aware of mispeculations, and at this cases it keeps a rename history in order to redo all the register renames and be able to redo the history with the appropriate registers.

Putting together the fact of not being on a loop, and having a memory-speculation with some cycles of delay between the mispeculation and the detection of that fact, implies that our current algorithm should take into account that fact to avoid misalignment of the vectorial instructions, because it is possible that on the new instruction-execution order our vectorial-candidates instructions are no longer candidates, and our renaming it isn't aware of this.
Possibilities around solving these problems have been studied but given the amount of time already spent on project and all the major changes that it could imply, adding some code major changes in some pipeline stages, and having to add changes on unchanged stages (commit stage, for instance), it was considered out of the scope of this project.

6. Related work

6.1 Speculative dynamic vectorization

Previously at this work an speculative dynamic vectorization system was developed by PhD Alex Pajuelo, Antonio González and Mateo Valero.

His work is based on the fact that when multiple instances of a load are repeated, usually the accessed memory address are based on adding some constraint to a base address, so it can be predicted. Based on that fact when a load is detected matching this pattern 2 or more times, they launch some vectorial operations getting the next data addresses, so they speculate that future loads would be done matching the pattern and vectorial registers are filled with the obtained data.

Then, when an arithmetic instruction is using some of loaded data a vectorial speculative version of the instruction is also generated, so speculating that future arithmetic instructions will enter on pipeline using the rest of the loaded data.

The scalar versions of instructions are reconverted into validation instructions, they validate that no store in between changed the loaded data, and that predicted addresses are those actually used by scalar instruction.

Their microarchitecture extension applied in a 4-way issue superscalar processor with one wide bus were 19% faster than the same processor with 4 scalar buses to L1 data cache.

This work is similar on our work, but the main design difference is that our work makes no speculation at all. It is based on real instructions, so there is no chance of invalid data. Main goal is also different, while his goal was to achieve better performance, on our work we knew that probably we would be losing performance, our goal was to save energy without losing too much performance. On conclusions we analyze the achievement of this.

6.2 CRIB: Consolidated Rename, Issue and Bypass

PhD Erika Guandi on 2010 at her doctoral thesis designed a modification of rename algorithm in order to save energy as our work tries to do.

Her work noticed that a large percentage of chip power is spent on operand delivery. The power consumed by operand delivery is even larger than the power needed for the execution itself. Thus, she proposed reconstructing the way out-of-order execution is done in order to eliminate power consumption by operand delivery.

She proposed CRIB: Consolidated Rename, Issue and Bypass as a solution to the power problem. Using CRIB, out-of-order is done without explicit register renaming. By removing explicit register renaming, several supporting structure needed for operand delivery can be eliminated. Hence, power consumption related to operand delivery can be dramatically reduced. CRIB also removes separation of data and dependency linking used in conventional out-of-order machine, resulting in the removal of speculative scheduling.

With removal of speculative scheduling, various cache optimizations that were not

attractive for conventional machine due to the added latency non-determinism can be employed.

Her detailed energy evaluation indicated that the average energy saving in the execution core for SPEC CPU 2006 integer and floating-point were 72% and 67% respectively compared to a conventional out-of-order machine with a large instruction window. Adding cache optimizations further increased the energy saving to 80% and 77% respectively.

6.3 VLIW processor architecture

VLIW stands for Very Long Instruction Word. It is a computer architecture where several instructions are executed at the same time in parallel, executing instructions in the scheduled order established by the compiler.

The motivation is to reduce the hardware complexity added in the out-of-order architectures to remove data hazards, which can led to potentially poor performance due to stalling pipeline because of hazards. To compensate the loss of performance due to execution in order several branches are added to execute multiple instructions at the same time.

Since in this architecture the instructions are in order, hardware complexity introduced by out-of-order architectures is removed. However, complexity is moved to compiler. Now compiler have to manage program code to schedule properly the instructions to execute at the same time in order to avoid misusing of processor performance, it has to determine which instructions can be done in parallel and transform them into a VLIW instruction.

This can only be achieved with programmer aid, in this architecture programmers have to code their algorithms in a way to help compiler schedule the instructions properly to maximize processors performance.

The drawback is that in this architecture scheme portability can not be maintained since programmers have to recode algorithms and recompile them to achieve better scheduling for a VLIW architecture. Notice the fact that compilers having to code binaries with VLIW instructions implies that backward compatibility with older binaries is still not possible even accepting the fact of performance lost, since instructions are not compatible.

7. Planning and costs

Development of proposed design was divided in several tasks.

1. **Analysis of the initial proposal design**. It consisted of studying the difficulties of the initial proposed design, searching failures and solving them to transform the initial design to a real viable proposal design idea. Some of this analysis can be shown in chapter 2 and sections 3.1 and 3.3.

2. **Designing the rename algorithm**. It was the study of which additional structures in the processors microarchitecture will be needed. This is described in section 3.2.

3. **Designing the dispatch stage**. As previous task, it consist of studying wich additional structures were needed. This is described in section 3.4.

4. **Studying gem5 simulator**. It was the study of how gem5 works in order to be able to implement proposal design into simulator. It is described in section 4.1.

5. **Implementation of rename algorithm**. Corresponds to section 4.2.

6. **Implementation of dispatch algorithm**. Corresponds to section 4.3.

7. **Validation and Evaluation**. Validation of design implementation and evaluation of it's performance. Described in chapter 5.

8. **Documentation**. This document elaboration.

The following picture depicts planned development times and dependencies between tasks.

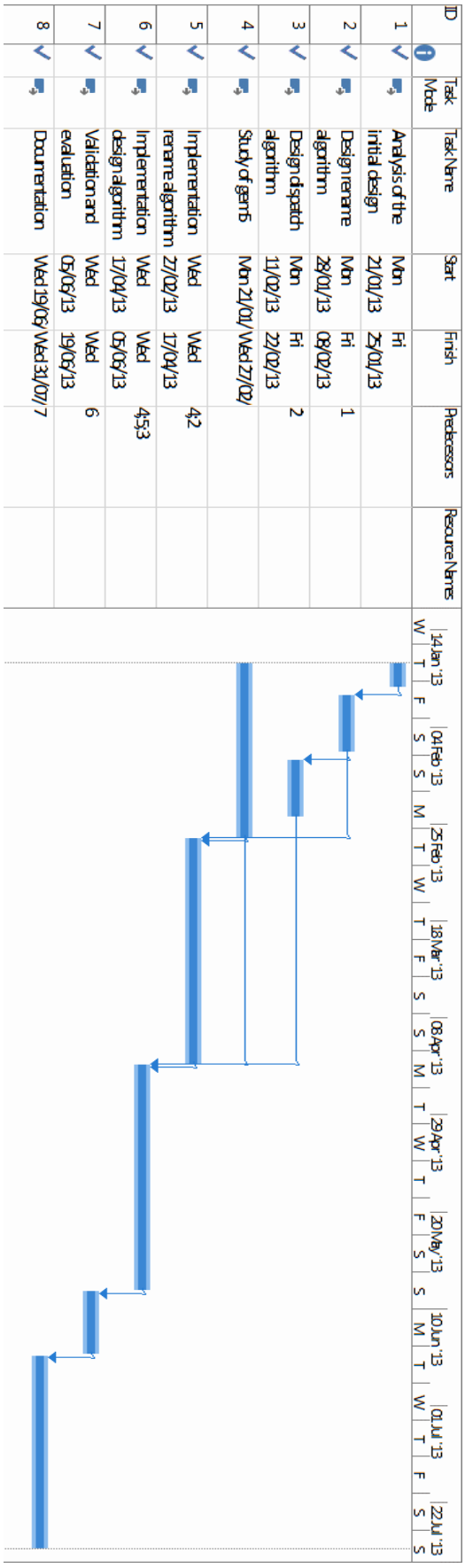


Figure 26: Tasks dependencies and planned hours

7.1 Development planned times

The following table summarizes the planned times for each task.

Task	Analysis & Design	Study of gem5	Implementation	Evaluation	Documentation	
1	20 hours					
2	40 hours					
3	40 hours					
4		110 hours				
5			140 hours			
6			140 hours			
7				40 hours		
8					120 hours	
Total	100 hours	110 hours	280 hours	40 hours	120 hours	**650 hours**

Figure 27: Planned time per task

7.2 Current development times

Following tables describes the real times that took each task to develop.

Task	Analysis & Design	Study of gem5	Implementation	Evaluation	Documentation	
1	20 hours					
2	40 hours					
3	40 hours					
4		140 hours				
5			150 hours			
6			200 hours			
7				40 hours		
8					120 hours	
Total	100 hours	140 hours	350 hours	40 hours	120 hours	**750 hours**

Figure 28: Real time per task

As it can be seen there is an important deviation in the scheduled plans during implementation. Initially it was believed that open-community mailing lists will help on the implementation, compensating the lack of documentation of gem5 simulator. Finally, community helped but it wasn't enough to solve some serious problems.

Fortunately investigating further into some conferences that took place around the world explaining the simulator and deeping further in source code, problems were solved and development could be finished successfully.

7.3 Costs

Consider now the cost that this project would have outside an academic environment. The following table summarizes human resources value:

Concept	Hours	Cost per hour	Total
Analysis	20	36 €	720 €
Design	80	30 €	2400 €
Implementation	490	24 €	11760 €
Evaluation	40	24 €	960 €
Documentation	120	24 €	2880 €
			18720 €

Figure 29: Human resources value

The following table summarizes the cost of hardware and software used for this project:

Concept	Value
Gem5 Simulator	Free
VirtualBox	Free
Kubuntu 10.04 LTS	Free
GCC Compiler	Free
Python compiler	Free
Building tools (Scons)	Free
IDE (Eclipse)	Free
Text editor (Vi iMproved)	Free
Laptop	832 €
Main desktop computer	1340 €
Total	**2172 €**

Figure 30: Hardware and software costs

The total cost of the project would have been of **20892 €**.

8. Conclusions and future work

Our project achieved the goal of vectorize instructions that a compiler could not vectorize, and in vectorial loops it vectorizes all possible instructions. Therefore the goal of saving energy through vectorization is achieved, since we are reducing the number of instructions executed – for each vectorized instruction, four scalar instructions are not executed –. [19]

However the overall performance (measured in CPI) is absolutely excessive and impossible to accept in a processor. This is due it was not thought that stalling some instructions will carry on successive stallings massively and, at the end, stalling the whole pipeline.

Therefore future work on this algorithm will have to proceed only stalling instructions belonging to loops. But this would probably not be enough, since loops can be very large and with lots of non-vectorial instructions. So it would have to be improved with some kind of detection mechanism to check if instructions use some data calculated in previous iterations. That could be done in hardware terms by adding some structures to store which memory positions are being produced and consumed between iterations, and therefore stalling only those instructions who do not consume data produced in earlier iterations. If that gets achieved and forcefully stalled instructions gets reduced to those who are good candidates to be vectorial ones, processor pipeline will probably not get stuck.

On the other hand, our second algorithm promises to be able to vectorize more instructions, since it does not depend on being inside some loop to vectorize instructions. However, the memory mispeculations are a problem due to misaligned positions. Therefore future work on this path would need to take into account those mispeculations and recalculate if instructions are vectorial candidates or not.

The second algorithm also will have the same performance bottleneck as the first one, stalling instructions which are not really vectorial. Consequently it also would be needed to apply the improvements on detecting good instruction candidates to be vectorized. But since it does not need loops, at a first glance it seems a better option to really improve performance.

9. References

[1] A. Pajuelo, A. González, and M. Valero. Speculative Dynamic Vectorization. In Proceedings of the 29th Annual International Symposium on Computer Architecture, May 2002.

[2] A. González, F. Latorre and G. Magklis. Processor Microarchitecture: An Implementation Perspective. 2011.

[3] Intel Corporation. A Guide to Vectorization with Intel C++ Compilers. 2012.

[4] Z. Zhang. Scoreboarding and Tomasulo Algorithm. Computer Systems Architecture 581. 2005.

[5] D. Patterson. Tomasulo Algorithm and Dynamic Branch Prediction. 1996.

[6] http://gem5.org. The gem5 Simulator System. 2013.

[7] E. Gunadi. CRIB: Consolidated Rename, Issue and Bypass. University of Wisconsin-Madison. 2010.

[8] J. Fisher. Very Long Instruction Word architectures and the ELI-152. Proceedings of the 10th annual international symposium on Computer architecture. International Symposium on Computer Architecture. 1983.

[9] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. IEEE International Symposium on Performance Analysis of Systems & Software. 2007.

[10] M. T. Chapman. Solutions for a more energy-efficient data center: from water to fresh-air cooling. IBM Symposium “Raising the thermal limits of your data center can lower your energy costs”, 2012.

[11] R. Stallman among others. The GNU C Reference Manual. 2011.

[12] https://docs.python.org/2/reference/. The Python Reference Manual. 2013.

[13] http://www.scons.org/. Scons manual. 2013.

[14] http://docs.nvidia.com/cuda/thrust/. Nvidia CUDA Toolkit Documentation. 2013.

[15] Advanced Micro Devices. 3DNow! Technology Manual. 2000.

[16] International Business Machines. PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual. 2006.

[17] S. Siewert. Using Intel Streaming SIMD Extensions and Intel Integrated Performance Primitives to Accelerate Algorithms. 2009.

[18] Intel Corporation. Intel C++ Compiler for Linux Intrinsics Reference. 2006.

[19] Juan M. Cebrian, Lasse Natvig, Jan Christian Meyer, "Improving Energy Efficiency through Parallelization and Vectorization on Intel Core i5 and i7 Processors," sccompanion, pp.675-684, 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, 2012.

Appendix A

A.1 Benchmark A

A.1.1 Rename debug trace

```
7762500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.0
7762500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.0
7762500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.0
7762500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.0
7763000: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.1
7763000: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.1
7763000: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.1
7763000: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.1
7763500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.2
7763500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.2
7763500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.2
7763500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.2
7764000: system.cpu.rename: PC: 0x00008986   adds   r2, r2, #1 C4 completat, assigno R0.3
7764000: system.cpu.rename: PC: 0x00008984   ldr   r0, [r4, r3] C4 completat, assigno R1.3
7764000: system.cpu.rename: PC: 0x00008988   ldr   r1, [r6, r3] C4 completat, assigno R2.3
7764000: system.cpu.rename: PC: 0x0000898e   adds   r3, r3, #4 C4 completat, assigno R3.3
7764500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.0
7764500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.0
7764500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.0
7764500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.0
7765000: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.1
7765000: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.1
7765000: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.1
7765000: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.1
7765500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.2
7765500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.2
7765500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.2
7765500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.2
7766000: system.cpu.rename: PC: 0x00008986   adds   r2, r2, #1 C4 completat, assigno R0.3
7766000: system.cpu.rename: PC: 0x00008984   ldr   r0, [r4, r3] C4 completat, assigno R1.3
7766000: system.cpu.rename: PC: 0x00008988   ldr   r1, [r6, r3] C4 completat, assigno R2.3
7766000: system.cpu.rename: PC: 0x0000898e   adds   r3, r3, #4 C4 completat, assigno R3.3
7766500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.0
7766500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.0
7766500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.0
7766500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.0
7767000: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.1
7767000: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.1
7767000: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.1
7767000: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.1
7767500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.2
7767500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.2
7767500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.2
7767500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.2
7768000: system.cpu.rename: PC: 0x00008986   adds   r2, r2, #1 C4 completat, assigno R0.3
7768000: system.cpu.rename: PC: 0x00008984   ldr   r0, [r4, r3] C4 completat, assigno R1.3
7768000: system.cpu.rename: PC: 0x00008988   ldr   r1, [r6, r3] C4 completat, assigno R2.3
7768000: system.cpu.rename: PC: 0x0000898e   adds   r3, r3, #4 C4 completat, assigno R3.3
7768500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.0
7768500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.0
7768500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.0
7768500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.0
7769000: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.1
7769000: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.1
7769000: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.1
7769000: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.1
7769500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.2
7769500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.2
7769500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.2
7769500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.2
7770000: system.cpu.rename: PC: 0x00008986   adds   r2, r2, #1 C4 completat, assigno R0.3
7770000: system.cpu.rename: PC: 0x00008984   ldr   r0, [r4, r3] C4 completat, assigno R1.3
7770000: system.cpu.rename: PC: 0x00008988   ldr   r1, [r6, r3] C4 completat, assigno R2.3
```

```
7770000: system.cpu.rename: PC: 0x0000898e   adds   r3, r3, #4 C4 completat, assigno R3.3
7770500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.0
7770500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.0
7770500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.0
7770500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.0
7771000: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.1
7771000: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.1
7771000: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.1
7771000: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.1
7771500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.2
7771500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.2
7771500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.2
7771500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.2
7772000: system.cpu.rename: PC: 0x00008986   adds   r2, r2, #1 C4 completat, assigno R0.3
7772000: system.cpu.rename: PC: 0x00008984   ldr   r0, [r4, r3] C4 completat, assigno R1.3
7772000: system.cpu.rename: PC: 0x00008988   ldr   r1, [r6, r3] C4 completat, assigno R2.3
7772000: system.cpu.rename: PC: 0x0000898e   adds   r3, r3, #4 C4 completat, assigno R3.3
7772500: system.cpu.rename: Assigno a PC: 0x00008986   adds   r2, r2, #1 el registre R0.0
7772500: system.cpu.rename: Assigno a PC: 0x00008984   ldr   r0, [r4, r3] el registre R1.0
7772500: system.cpu.rename: Assigno a PC: 0x00008988   ldr   r1, [r6, r3] el registre R2.0
7772500: system.cpu.rename: Assigno a PC: 0x0000898e   adds   r3, r3, #4 el registre R3.0
```

A.1.2 Dispatch debug trace

```
7762500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7762500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7762500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7763000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7763000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7763000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7763500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7763500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7763500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7764000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7764000: system.cpu.iq: New block of 4 instances completed
7764000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7764000: system.cpu.iq: New block of 4 instances completed
7764000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7764000: system.cpu.iq: New block of 4 instances completed
7764000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7764000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7764500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7764500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7764500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7764500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7764500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7765000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7765000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7765000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7765000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7765000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7765500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7765500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7765500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7765500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7765500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7766000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7766000: system.cpu.iq: New block of 4 instances completed
7766000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7766000: system.cpu.iq: New block of 4 instances completed
7766000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7766000: system.cpu.iq: New block of 4 instances completed
7766000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7766000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7766500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7766500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7766500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7766500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7766500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7767000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7767000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
```

```
7767000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7767000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7767000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7767500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7767500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7767500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7767500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7767500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7767500: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 10 cicles
7767500: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 9 cicles
7767500: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 8 cicles
7767500: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 7 cicles
7768000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7768000: system.cpu.iq: New block of 4 instances completed
7768000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7768000: system.cpu.iq: New block of 4 instances completed
7768000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7768000: system.cpu.iq: New block of 4 instances completed
7768000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7768000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7768500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7768500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7768500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7768500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7768500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7769000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7769000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7769000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7769000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7769000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7769500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7769500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7769500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7769500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7769500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7770000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7770000: system.cpu.iq: New block of 4 instances completed
7770000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7770000: system.cpu.iq: New block of 4 instances completed
7770000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7770000: system.cpu.iq: New block of 4 instances completed
7770000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7770000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7770000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 11 cicles
7770000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 10 cicles
7770000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 9 cicles
7770000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 8 cicles
7770500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7770500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7770500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7770500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7770500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7771000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7771000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7771000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7771000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7771000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7771500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7771500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7771500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7771500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7771500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7772000: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7772000: system.cpu.iq: New block of 4 instances completed
7772000: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
7772000: system.cpu.iq: New block of 4 instances completed
7772000: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7772000: system.cpu.iq: New block of 4 instances completed
7772000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7772000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7772500: system.cpu.iq: Stalling instruction PC 0x8986   adds   r2, r2, #1
7772500: system.cpu.iq: Stalling instruction PC 0x898a   adds   r1, r0, r1
```

```
7772500: system.cpu.iq: Stalling instruction PC 0x898e   adds   r3, r3, #4
7772500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7772500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7773000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7773000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7773000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 13 cicles
7773000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 12 cicles
7773000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 11 cicles
7773000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 10 cicles
7773500: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7773500: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7774000: system.cpu.iq: Trobo load amb productor bloquejat. Desbloquejar productor PC: 0x898e
7774000: system.cpu.iq: Esborro   adds   r3, r3, #4 per desbloquejar load
7776000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 15 cicles
7776000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 14 cicles
7776000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 13 cicles
7776000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 12 cicles
7779000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 17 cicles
7779000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 16 cicles
7779000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 15 cicles
7779000: system.cpu.iq: Allibero PC: 0x898a   adds   r1, r0, r1 despres de 14 cicles
7792000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 3 cicles
7792000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 2 cicles
7792000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 1 cicles
7792000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 56 cicles
7794000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 3 cicles
7794000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 2 cicles
7794000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 1 cicles
7794000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 56 cicles
7796000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 3 cicles
7796000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 2 cicles
7796000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 1 cicles
7796000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 56 cicles
7798000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 3 cicles
7798000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 2 cicles
7798000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 1 cicles
7798000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 56 cicles
7800000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 3 cicles
7800000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 2 cicles
7800000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 1 cicles
7800000: system.cpu.iq: Allibero PC: 0x8986   adds   r2, r2, #1 despres de 56 cicles
```

A.2 Benchmark B

A.2.1 Rename debug trace

```
7679500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.0
7680500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.0
7680500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.0
7681500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.1
7682500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.1
7682500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.1
7683500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.2
7684500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.2
7684500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.2
7685500: system.cpu.rename: PC: 0x0000897e   adds   r3, r3, #1 C4 completat, assigno R0.3
7686500: system.cpu.rename: PC: 0x00008980   ldr.w   r2, [r6, #4]! C4 completat, assigno R1.3
7686500: system.cpu.rename: PC: 0x0000897a   ldr.w   r1, [r4, #4]! C4 completat, assigno R2.3
7687500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.0
7688500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.0
7688500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.0
7689500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.1
7690500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.1
7690500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.1
7691500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.2
7692500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.2
7692500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.2
7693500: system.cpu.rename: PC: 0x0000897e   adds   r3, r3, #1 C4 completat, assigno R0.3
7694500: system.cpu.rename: PC: 0x00008980   ldr.w   r2, [r6, #4]! C4 completat, assigno R1.3
7694500: system.cpu.rename: PC: 0x0000897a   ldr.w   r1, [r4, #4]! C4 completat, assigno R2.3
```

```
7695500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.0
7696500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.0
7696500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.0
7697500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.1
7698500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.1
7698500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.1
7699500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.2
7700500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.2
7700500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.2
7701500: system.cpu.rename: PC: 0x0000897e   adds   r3, r3, #1 C4 completat, assigno R0.3
7702500: system.cpu.rename: PC: 0x00008980   ldr.w   r2, [r6, #4]! C4 completat, assigno R1.3
7702500: system.cpu.rename: PC: 0x0000897a   ldr.w   r1, [r4, #4]! C4 completat, assigno R2.3
7703500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.0
7704500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.0
7704500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.0
7705500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.1
7706500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.1
7706500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.1
7707500: system.cpu.rename: Assigno a PC: 0x0000897e   adds   r3, r3, #1 el registre R0.2
7708500: system.cpu.rename: Assigno a PC: 0x00008980   ldr.w   r2, [r6, #4]! el registre R1.2
7708500: system.cpu.rename: Assigno a PC: 0x0000897a   ldr.w   r1, [r4, #4]! el registre R2.2
7709500: system.cpu.rename: PC: 0x0000897e   adds   r3, r3, #1 C4 completat, assigno R0.3
```

A.2.2 Dispatch debug trace

```
7679500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7680500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7681500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7682500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7683500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7684500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7685500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7685500: system.cpu.iq: New block of 4 instances completed
7686500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7686500: system.cpu.iq: New block of 4 instances completed
7687500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7688500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7689500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7690500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7691500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7692500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7693500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7693500: system.cpu.iq: New block of 4 instances completed
7694500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7694500: system.cpu.iq: New block of 4 instances completed
7695500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7696500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7697500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7698500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7699500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7700500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7701500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7701500: system.cpu.iq: New block of 4 instances completed
7702500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7702500: system.cpu.iq: New block of 4 instances completed
7703500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7704500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7705500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7706500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7707500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7708500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7709500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7709500: system.cpu.iq: New block of 4 instances completed
7710500: system.cpu.iq: Stalling instruction PC 0x8986   add   r2, r2, r1
7710500: system.cpu.iq: New block of 4 instances completed
7711500: system.cpu.iq: Stalling instruction PC 0x897e   adds   r3, r3, #1
7713500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 12 cicles
7713500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 8 cicles
7713500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 4 cicles
7713500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 56 cicles
7714500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 12 cicles
```

```
7714500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 8 cicles
7714500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 4 cicles
7714500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 56 cicles
7721500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 12 cicles
7721500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 8 cicles
7721500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 4 cicles
7721500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 56 cicles
7722500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 12 cicles
7722500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 8 cicles
7722500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 4 cicles
7722500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 56 cicles
7729500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 12 cicles
7729500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 8 cicles
7729500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 4 cicles
7729500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 56 cicles
7730500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 12 cicles
7730500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 8 cicles
7730500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 4 cicles
7730500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 56 cicles
7737500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 12 cicles
7737500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 8 cicles
7737500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 4 cicles
7737500: system.cpu.iq: Allibero PC: 0x897e   adds   r3, r3, #1 despres de 56 cicles
7738500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 12 cicles
7738500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 8 cicles
7738500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 4 cicles
7738500: system.cpu.iq: Allibero PC: 0x8986   add   r2, r2, r1 despres de 56 cicles
```

A.3 Benchmark C

A.3.1 Rename debug trace

```
7869500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7869500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7869500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7870500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7870500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7870500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7871500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7871500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7871500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7872500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7872500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7872500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7873500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7873500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7873500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7874500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7874500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7874500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7875500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7875500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7875500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7876500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7876500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7876500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7877500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7877500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7877500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7884000: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7884000: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7884500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7884500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7885500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7885500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7885500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7886500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7886500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7886500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7887500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
```

```
7887500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7887500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7888500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7888500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7888500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7889500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7889500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7889500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7890500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7890500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7890500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7891500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7891500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7891500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7892500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7892500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7892500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7893500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7893500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7893500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7894500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7894500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7894500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7895500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7895500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7895500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7896500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7896500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7896500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7897500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7897500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7897500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7898500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7898500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7898500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7899500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7899500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7899500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7900500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7900500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7900500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7901500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7901500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7901500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7902500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7902500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7902500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7903500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7903500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7903500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7904500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7904500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7904500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7905500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7905500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7905500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7906500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7906500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7906500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7907500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7907500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7907500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7908500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7908500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7908500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7909500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7909500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7909500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7910500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7910500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7910500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7911500: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
```

```
7911500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7911500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7912500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7912500: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7912500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7913500: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7913500: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7913500: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7936500: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7937000: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7937000: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
7938000: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.2
7938000: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.1
7938000: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.2
7939000: system.cpu.rename: PC: 0x00008a28   adds   r3, r3, #1 C4 completat, assigno R0.3
7939000: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.2
7939000: system.cpu.rename: PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] C4 completat, assigno R2.3
7940000: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.0
7940000: system.cpu.rename: PC: 0x00008a2a   ldr   r1, [r2, #0] C4 completat, assigno R1.3
7940000: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.0
7941000: system.cpu.rename: Assigno a PC: 0x00008a28   adds   r3, r3, #1 el registre R0.1
7941000: system.cpu.rename: Assigno a PC: 0x00008a2a   ldr   r1, [r2, #0] el registre R1.0
7941000: system.cpu.rename: Assigno a PC: 0x00008a2e   ldr.w   r0, [r4, -r0 LSL #2] el registre R2.1
```

A.3.2 Dispatch debug trace

```
7869500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7869500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7870500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7870500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7871500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7871500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7872500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7872500: system.cpu.iq: New block of 4 instances completed
7872500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7872500: system.cpu.iq: New block of 4 instances completed
7873500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7873500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7874500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7874500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7875500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7875500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7876500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7876500: system.cpu.iq: New block of 4 instances completed
7876500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7876500: system.cpu.iq: New block of 4 instances completed
7876500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 14 cicles
7876500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 12 cicles
7876500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 10 cicles
7876500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 8 cicles
7877500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7877500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7884000: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7884500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7885500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7885500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7885500: system.cpu.iq: New block of 4 instances completed
7886500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7886500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7887500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7887500: system.cpu.iq: New block of 4 instances completed
7887500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7888500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7888500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7889500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7889500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7889500: system.cpu.iq: New block of 4 instances completed
7890500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7890500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7891500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7891500: system.cpu.iq: New block of 4 instances completed
```

```
7891500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7892500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7892500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7893500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7893500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7893500: system.cpu.iq: New block of 4 instances completed
7894500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7894500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7895500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7895500: system.cpu.iq: New block of 4 instances completed
7895500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7896500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7896500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7897500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7897500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7897500: system.cpu.iq: New block of 4 instances completed
7898500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7898500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7899500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7899500: system.cpu.iq: New block of 4 instances completed
7899500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7900500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7900500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7900500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 6 cicles
7900500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 4 cicles
7900500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7900500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 56 cicles
7901500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7901500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7901500: system.cpu.iq: New block of 4 instances completed
7902500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7902500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7903500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7903500: system.cpu.iq: New block of 4 instances completed
7903500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7904500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7904500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7904500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 6 cicles
7904500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 4 cicles
7904500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7904500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 56 cicles
7904500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 6 cicles
7904500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 4 cicles
7904500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 2 cicles
7904500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 56 cicles
7905500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7905500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7905500: system.cpu.iq: New block of 4 instances completed
7906500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7906500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7907500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7907500: system.cpu.iq: New block of 4 instances completed
7907500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7908500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7908500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7909500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7909500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7909500: system.cpu.iq: New block of 4 instances completed
7910500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7910500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7911500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7911500: system.cpu.iq: New block of 4 instances completed
7911500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7912500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7912500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7913500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7913500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7913500: system.cpu.iq: New block of 4 instances completed
7913500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 16 cicles
7913500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 3 cicles
7913500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 2 cicles
7913500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 56 cicles
```

```
7914500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7914500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7915500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 20 cicles
7915500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 4 cicles
7915500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7915500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 56 cicles
7917500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 6 cicles
7917500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 4 cicles
7917500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 2 cicles
7917500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 56 cicles
7919500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 6 cicles
7919500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 4 cicles
7919500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7919500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 56 cicles
7921500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 6 cicles
7921500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 4 cicles
7921500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 2 cicles
7921500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 56 cicles
7923500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 6 cicles
7923500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 4 cicles
7923500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7923500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 56 cicles
7925500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 6 cicles
7925500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 4 cicles
7925500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 2 cicles
7925500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 56 cicle
7927500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 6 cicles
7927500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 4 cicles
7927500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7927500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 56 cicles
7929500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 6 cicles
7929500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 4 cicles
7929500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 2 cicles
7929500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 56 cicles
7931500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 6 cicles
7931500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 4 cicles
7931500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7931500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 56 cicles
7933500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 6 cicles
7933500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 4 cicles
7933500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 2 cicles
7933500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 56 cicles
7935500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 6 cicles
7935500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 4 cicles
7935500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7935500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 56 cicles
7937500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 6 cicles
7937500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 4 cicles
7937500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 2 cicles
7937500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 56 cicles
7939000: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7939500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7939500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 6 cicles
7939500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 4 cicles
7939500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7939500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 56 cicles
7940500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7940500: system.cpu.iq: New block of 4 instances completed
7940500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7940500: system.cpu.iq: New block of 4 instances completed
7941500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7941500: system.cpu.iq: Stalling instruction PC 0x8a32   add   r1, r1, r0
7941500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 6 cicles
7941500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 4 cicles
7941500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 2 cicles
7941500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 56 cicles
7941500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7941500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 54 cicles
7941500: system.cpu.iq: Allibero PC: 0x8a28   adds   r3, r3, #1 despres de 2 cicles
7942500: system.cpu.iq: Stalling instruction PC 0x8a28   adds   r3, r3, #1
7968500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 52 cicles
7968500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 3 cicles
```

```
7968500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 2 cicles
7968500: system.cpu.iq: Allibero PC: 0x8a32   add   r1, r1, r0 despres de 56 cicles
```