Advanced Computer Architectures

Niccoló Didoni

March 2022

Contents

Part I

Branch prediction

Chapter 1

Static branch prediction

1.1 Introduction

Branch prediction is the action of predicting if a branch instruction is going to take a branch (i.e. is going to jump to another part in the code) or to continue execution without jumping. Branch prediction is fundamental to improve a processor's performance.

The easiest method for predicting the behaviour of a program is static branch prediction. Static branch prediction is executed by a compiler that at compile-time decides if a certain branch should be taken or not. Since this type of branch prediction is static, once the compiler has decided, for each branch, the next instruction to be executed, the prediction is kept until the program is recompiled.

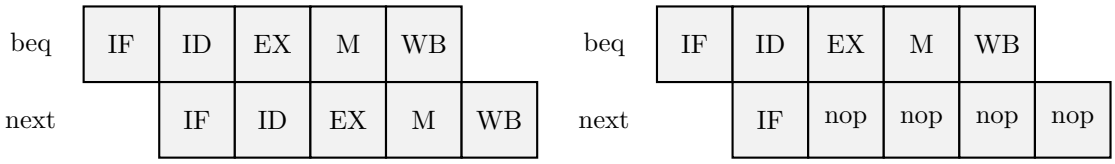
Program correctness When predicting branches we have to be sure that the outcome of the program is correct, whether the prediction is correct or not. To ensure that the correct instruction is always executed, when a prediction is wrong, the next operations (the next operation in MIPS) have to be flushed transforming the instructions in nops.

Profiling Branch prediction is usually applied when the application is highly predictable, i.e. when from static analysis we can define the behaviour of a program with good precision. Usually we use profiling to determine, at design-time, the behaviour of a program. This technique predict the behaviour of a program running multiple simulation of different data-sets.

1.2 Static analysis techniques

In MIPS, branch prediction is done in the Instruction Decode (ID) stage, thus everything about the prediction should be ready before the ID stage. This means that at the end of the Instruction Fetch (IF) stage the prediction has to be ready so that the ID stage can check if the prediction is correct and prepare the address for the jump. In particular, in the ID stage

- If the prediction is correct, execution continues normally. The behaviour of a program when the prediction is correct is shown in Figure 1.1a.
- If the prediction is wrong, the processor has to flush all operations started after the branch instruction. Since the prediction is checked in the ID stage, only one instruction has to be



(a) Correct branch prediction.

(b) Wrong branch prediction.

Figure 1.1: Instructions' behaviours for branch prediction in the general case.

flushed (i.e. the instruction that has just executed the IF stage). This behaviour is shown in Figure 1.1b.

There exists 5 techniques to do static branch prediction,

- **Branch always not taken**.
- **Branch always taken**.
- **Backward taken forward not taken**.
- **Profile driven branch prediction**.
- **Delayed branch prediction**.

1.2.1 Branch always not taken

The branch always not taken technique is the easiest one because the processor simply fetches the next instruction (i.e. predicts that the branch is never taken). This means that the processor doesn't have to compute the Branch Target Address (BTA).

If the prediction is wrong, we only lose one clock cycle because we flush only one instruction (i.e. the one executed after the branch). In other words this technique has a **1 cycle penalty** for wrong predictions.

Usage The branch always not taken technique is very effective when we execute if-then-else blocks in which one of the blocks is usually taken (e.g. the `then` block is more probable to be executed).

1.2.2 Branch always taken

The branch always not taken technique is the dual of the branch always taken. In this case the processor assumes that the branch should always be taken. This technique has some critic points, to analyse them let us consider a while loop

```
while (cond) {
    /* code */
}
```

The code can be translated in MIPS as in Figure 1.2.

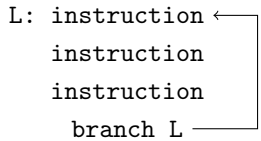


Figure 1.2: MIPS translation of a while loop.

Branch Target Address buffer

A key information in this technique is the BTA, that is the address to which the processor has to jump to. This address is stored in a buffer in a cache. The cache ideally contains the buffers for every branch instruction of the process but in practice it's limited by its size, thus the processor has to override some addresses when there are more branch instructions than buffer slots in the cache. Let us consider the ideal cache. In any case, the processor checks if a buffer is associated to the branch instruction that it's currently executed to avoid fetching a wrong instruction (i.e. an instruction of another branch).

Once the BTA for a branch is buffered, it can be used each time that branch is executed. The buffer is located in the IF stage, in fact the processor decides in this stage what instruction has to be executed next. Notice that we have to use a buffer because the computation of the branch address is computed in the ID stage.

Wrong predictions

As we can see from Figure 1.2, the first time the loop is executed, the processor hasn't computed the BTA yet, thus our prediction will be wrong. This happens because the buffer from which the target address is taken is outdated, thus it's different from the program counter (the comparison is done, as always, in the ID stage). The processor also guesses wrong when the last iteration of the loop is executed, in fact in this case it doesn't have to branch. To sum things up

1. At the first iteration the processor typically predicts a wrong address (because of the outdated buffer), thus it pays a 1 cycle penalty (the comparison is done in the ID stage, thus only the next instruction has to be flushed).

2. After the first iteration the buffer is updated and the processor keeps fetching the right instruction.

3. At the last iteration the processor wrongly predicts a branch and 1 instruction is stalled (for the same reason seen for the first iteration).

1.2.3 Backward taken forward not taken

The backward taken forward not taken technique merges the aforementioned techniques. In particular

- The processor predicts branch not taken for forward branches (i.e. branches in which we jump to a bigger address in the code). This choice comes from the fact that forward branches are usually used in `if-then-else`. In particular we can put the most probable path in the `then` block. This means that only a few times we have to take the branch, thus it's more probable to execute the `then` block, which is executed when the branch is not taken.

- The processor predicts branch taken for backward branches (i.e. branches in which we jump to a smaller address in the code). This choice comes from the fact that backward branches are used in while loops (like in Figure 1.2), thus the branch is taken most of the times.

1.2.4 Profile driven branch prediction

The profile driven branch prediction technique allows to analyse a program at design time to understand, for each branch, the probability to take a branch. Such analysis is done running the code multiple times with different data-sets and the predictions can be inserted in the code using compiler hints. This technique relies a lot on the compiler that has to use hints to predict the behaviour of a program.

1.2.5 Delayed branch prediction

The delayed branch prediction technique leverages the instruction after a branch. This operation, if the branch prediction is wrong, is usually flushed. If we replace the instruction with another instruction that is independent from the fact that the prediction is correct or wrong, we can avoid stalling the processor for one cycle and execute the independent instruction, even if the prediction is wrong. Basically the idea is to delay the fetching of the instruction after the branch instruction (BTA or next address) and replace it with an independent instruction. Notice that this techniques still uses prediction, but the advantage is that if the prediction is wrong, the processor doesn't have to stall because the instruction that should have been flushed is independent from the prediction and can be executed anyways. The candidate instruction can be searched inside or outside the loop body (or in the branch in cases of `if-then-else`). From now on, let us call **delay slot** the slot after the branch instruction.

The main problem with this technique is finding an instruction that is independent from the fact that the prediction is right or wrong. The candidate instruction can be searched

- **Form before**.
- **Form target**.
- **Form fall-through**.
- **Form after**.

To understand how such techniques work, let us divide a piece of code that contains a branch in four basic blocks (i.e. blocks of code that contain no jump or branch).

- The **before block** is executed before a branch.
- The **fall-through block** is executed if the branch is not taken.
- The **target block** is executed if the branch is taken.
- The **after block** is executed after the fall-through or the target block.

A visual representation of such blocks is shown in Figure 1.3 and Figure 1.4. Now that we know how to divide a the instruction around a branch we can specify the four techniques

- **From before**. The from before technique allows to pick an appropriate instruction from the before block.

```
    instruction
    instruction
     branch L
    instruction
    instruction
L:  instruction
    instruction
    instruction
    instruction
```

Figure 1.3: A piece of code with a branch instruction divided in basic code blocks.

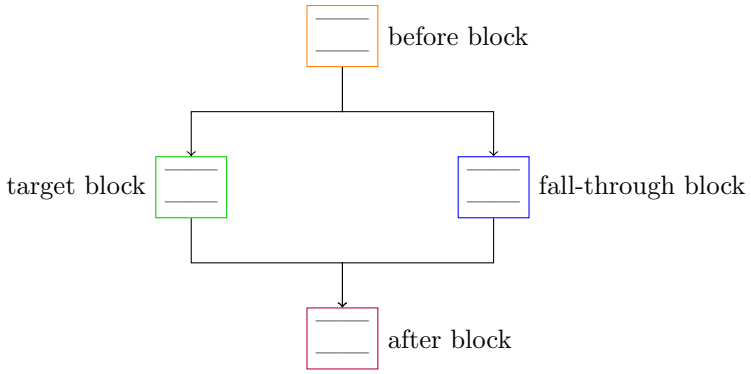


Figure 1.4: The blocks of a piece of code with a branch instruction.

- **From target**. The from before technique allows to pick an appropriate instruction from the target block.
- **From fall through**. The from fall-through technique allows to pick an appropriate instruction from the fall-through block.
- **From after**. The from after technique allows to pick an appropriate instruction from the after block.

In all these cases we have to remember that **no register in the candidate instruction should be used in the branch instruction**.

Limitations The main issues related to this technique are related to the compiler, in fact it has to be smart to correctly choose the instruction to execute in the delay slot. To help the compiler, some architectures provide an instruction called **cancel-branch** that flushes the instruction in the delay slot if the prediction is wrong. This helps the compiler choose an instruction with fewer constraints (in fact it doesn't have to ensure that the program is correctly executed when the prediction is wrong). Notice that even with the cancel-branch instruction, we still have to ensure that the behaviour of the program is correct.

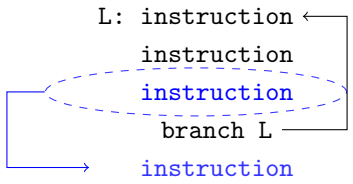


Figure 1.5: An instruction inside a loop is moved in the delay slot.

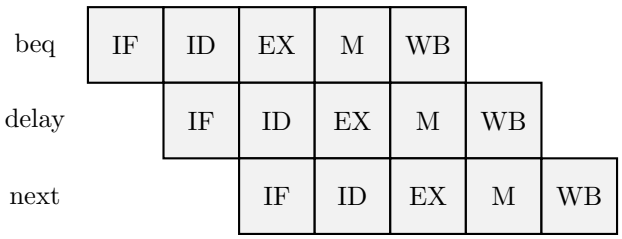


Figure 1.6: Instructions when using the delayed branch technique.

Chapter 2

Dynamic branch prediction

Dynamic branch prediction allows to dynamically predict the behaviour of a branch instruction using the past results. Predictions aren't done from software but are continuously updated by a dedicated hardware that remembers the past executions and predicts the future behaviour. Notice that even in this case the predictor can be wrong so we might have to pay a penalty. As always a MIPS processor pays only a one cycle penalty because if a prediction is wrong, when the process realises it (i.e. in the ID stage), only one instruction has entered the pipeline.

Hardware location The hardware for dynamic branch prediction is located in the IF stage, since the decision about the instruction to fetch is done in the IF stage. Remember that the prediction has to be done in the IF stage because it has to be checked in the ID stage (i.e. it has to be ready before the ID stage, thus in the IF stage).

In particular a dynamic branch predictor is made of two components

- A **Branch Outcome Predictor** (BOP). A BOP is a cache with buffers that store the past outcomes of the branches and returns, for each instruction, the prediction for each branch. Ideally the cache stores the history for all branches but in reality we have to overwrite some instructions because the cache has a limited space. When talking about predictions we are going to indicate the taken prediction as T and the not taken prediction as NT.
- A **Branch Address Buffer** (BAB) also called Branch Target Predictor. The BAB predicts, for each branch instruction, the address to which the processor has to jump. This address is called **Predicted Target Address** (PTA).

In a nutshell,

- The BOP predicts **if** we have to take a branch.
- The BAB predicts **where** we have to branch.

To better understand how these components work, let us consider the behaviour of a branch instruction in different situations.

Branch not taken Let us consider the case in which the hardware predicts branch not taken.

1. Initially, the BOP predicts branch not taken.

2. The processor fetches the next instruction (we don't need the PTA because we are not taking a branch) in the IF stage.
3. The processor confronts the prediction (i.e. the next instruction) with the program counter.
   - If the prediction is correct, the processor continues normally.
   - If the prediction is wrong, the processor has to flush an instruction (i.e. 1 cycle penalty).

Branch taken Let us consider now the case in which the hardware predicts branch taken.

1. Initially, the BOP predicts branch taken.
2. The BAB returns the predicted target address (PTA) in the IF stage and fetches the instruction at PTA.
3. The processor confronts the prediction (i.e. PTA) with the program counter.
   - If the prediction is correct, the processor continues normally.
   - If the prediction is wrong, the processor has to flush an instruction (i.e. 1 cycle penalty).

Classification of dynamic branch predictors A Branch Outcome Predictor can be implemented using

- A **branch history table**.
- A **correlating branch predictor**.
- A **two-level adaptive branch predictor**.

Instruction structure Before diving into the functioning of BABs and BOPs, let analyse the address of a branch instruction. An address A of length n can be divided in

- An **index** A_i of the k least-significant bits.
- A **tag** A_t of the $m = n - k$ most-significant bits.

This means that an address A can be written as

$$A = A_t.A_i$$

where the . (dot) notation represents concatenation. A graphical representation is shown in Figure 2.1.

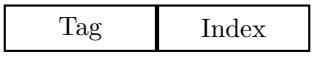


Figure 2.1: How a branch instruction address is divided.

2.1 Branch history table

The easiest way to implement a outcome predictor is to use a cache that stores the past predictions (i.e. an history of the predictions) for each branch. This history can be used to predict the future behaviour of a branch.

2.1.1 Single bit BHT

The simplest branch history table is a cache where each entry

- Has one bit to store the **last prediction** (i.e. each entry records the history of the immediately previous prediction).
- Is addressed by k bits (i.e. a key of the entry is k bits long), called **index**. This means that the table has 2^k entries in total.

When the processor wants to predict the behaviour of an instruction with address A (i.e. A is the address of the instruction itself), it takes the last k bits (i.e., the least-significant ones, i.e., the index) of A and uses them to take the value from the cache. In other words if we divide A in $A_m + A_k$, we can use A_k to get a value (i.e. the bit that represents the previous behaviour and the prediction) from the cache. An example of BHT is shown in Table 2.1.

k-bit address (index)	prediction (history)
0x0000	1
0x0001	0
⋮	⋮
0xfffe	1
0xffff	0

Table 2.1: A 1-bit Branch History Table.

Notice that in general k is smaller than the length of the address, thus we can't be sure that the prediction refers to the branch we are executing and which behaviour we want to predict. In other words, the predictor doesn't use a tag (i.e., the most significant bits A_m of the address) to check if the value retrieved refers to the correct address. For instance, the addresses `0b0110` and `0b1110` map to the same table for $k = 2$. This however isn't an issue (in terms of correctness of the program) because if the prediction is wrong, we can always stall the processor for one clock cycle.

Prediction correction

A BHT predicts the behaviour of a branch in the IF stage but, when the prediction is checked, we have to fix, if necessary, the last prediction. Basically we want to toggle the prediction if it's wrong. This concept can be expressed with the finite state automaton in Figure 2.2 (nodes represent the current status of the table, edges are the correct predictions).

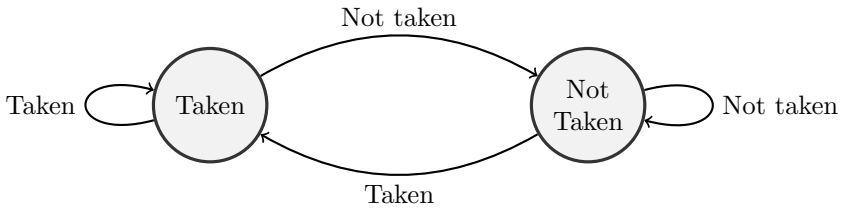


Figure 2.2: The finite state machine of BHT's bit.

Mispredictions

Branch history tables mispredict the behaviour of a branch in two situations

- When the behaviour changed with respect to the previous prediction (e.g. in a loop we hit the last iteration).
- When there is a conflict between two instructions. This problem can be solved adding more bits to the addresses of the cache or using hash functions over the full instruction address (i.e. we replace A_k with $hash(A)$).

We also have to consider the fact that when the processor is turned on, the BHT is initialised with random values, thus a misprediction might happen when first executing a branch.

Let us consider for instance a while loop. In this case we mispredict at the end of the loop and at the beginning of the next iteration because in both cases the behaviour changes with respect to the previous iteration (if we assume no instruction conflict). For this reason we can define the accuracy of a BHT as

$$accuracy = \frac{\#misses}{\#iterations} = \frac{2}{\#iterations}$$

2.1.2 2-bit BHT

To improve accuracy, we can increment the number of bits used in the buffer. In particular if we use two bits we obtain a finite state machine in Figure 2.3. Basically in this case we are keeping a longer history.

The improved accuracy of this model comes from the fact that the prediction has to be wrong twice before changing the prediction. If we consider the same example we did for 1-bit BHT we obtain an accuracy of

$$accuracy = \frac{\#misses}{\#iterations} = \frac{1}{\#iterations}$$

because the processor stalls only at the first iteration.

Multiple bit BHT

Theoretically the buffer can be expanded to n bits. This architecture is called **saturating counter**. Studies demonstrate however that 2 bits are enough to get a performance similar to a general n-bit BHT. This means that it's better to improve k for better performances.

2.2 Correlating branch predictor

Branch history tables used the history of a single branch to predict the behaviour of the same branch. Correlating Branch Predictors (CBP) make use of multiple branch history tables and predict the behaviour of a branch using the global history (i.e. not the local history of a branch like in BHTs). In particular, each correlating branch predictor is made of

- A **buffer** that stores the global history of the program (i.e. the last m branch results).
- 2^m branch history tables that work exactly as before (i.e. with n bits, k addresses and 2^k rows).

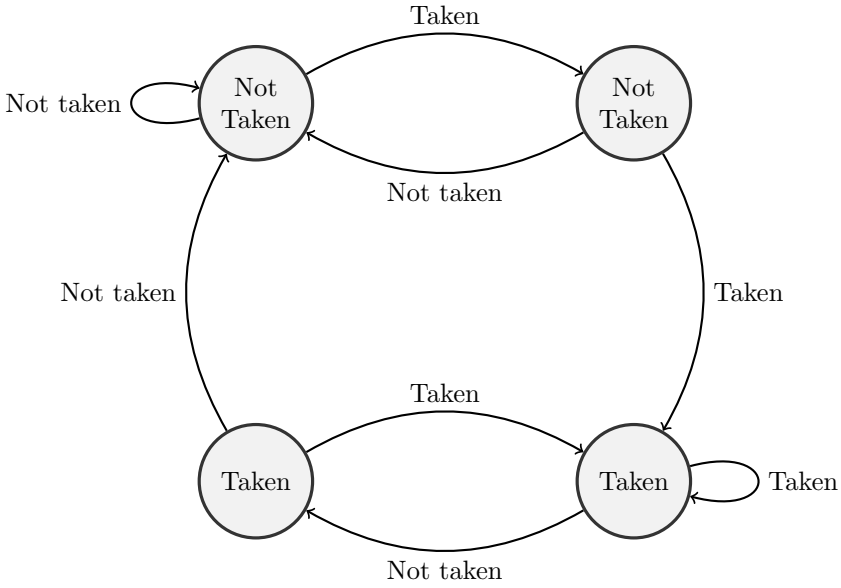


Figure 2.3: The finite state machine of BHT's buffer with two bits.

- A **multiplexer**. The multiplexer uses the previous history to select from which branch history table we should take the prediction.

In general a CBP can have n bits for the BHT and m bits for the global history buffer. Since the history buffer has m bits, the multiplexer can choose among 2^m inputs, thus we can have 2^m different BHTs.

Usage This family of predictors is very useful when we have nested loops of control sequences nested in a loop.

2.2.1 (1, 1)-correlating branch predictors

Let us analyse the simplest branch predictor first a CBP(1, 1), that is a predictor that stores only the previous branch prediction in the global history and for which each BHT has only one bit. In this case

- One BHT stores the prediction for branch not taken (i.e. a prediction is taken from this table if the previous history contains a not taken result).
- One BHT stores the prediction for branch taken (i.e. a prediction is taken from this table if the previous history contains a taken result).

2.2.2 (2, 2)-correlating branch predictors

As for BHTs we can expand the basic (1, 1) architecture and add more BHTs and bits in the BHT. In particular, let us consider a CBP(2, 2) that has

- A 2-bits global history buffer.

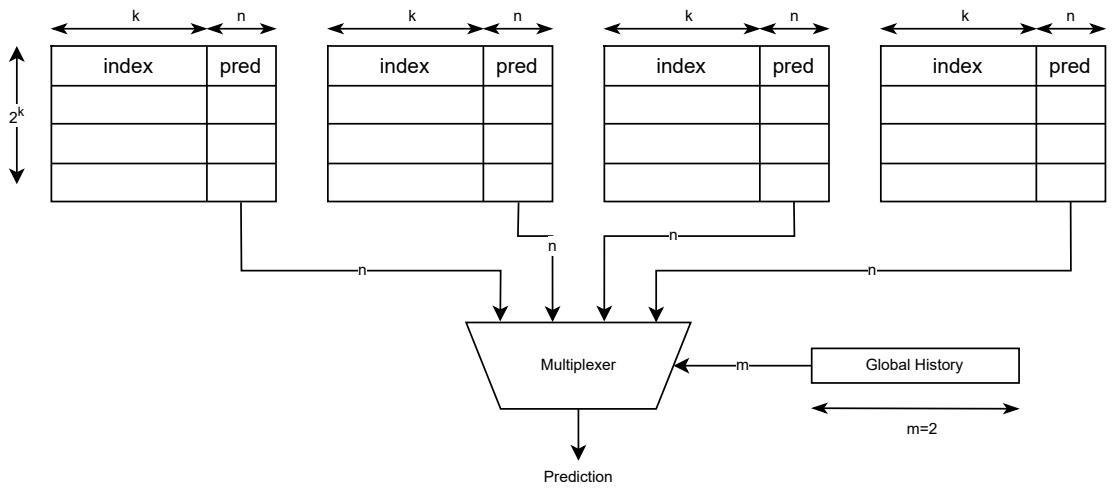


Figure 2.4: A (n,m)-correlating branch predictor.

- $2^2 = 4$ BHTs each of which with 2 bits.

In this case each BHT represents a combination of the global history, in particular (T, T), (T, NT), (NT, T), (NT, NT).

Performance Even for CBPs it has been empirically demonstrated that a CBP(2, 2) has similar performances of a general CBP, thus it's not convenient to add more BHTs or more bits to each BHT. We can instead increase the number of bits k used for addressing.

If we compare CBPs with BHTs, we can say that the former outperform the latter even with unlimited number of entries (i.e. $k = \infty$). This means that in general a CBP is better than a BHT.

BHTs as a special case of CBPs Notice that a branch history table can be seen as a correlating branch predictor that has 0 history bits (thus only one BHT).

$$BHT(n) \equiv CBP(0, n)$$

2.3 Two-level adaptive branch predictor

Two-Level Adaptive Branch Predictors (2LABP) are completely different with respect to the previous solutions. In this case the processor has

- A **Branch History Register** (BHR), i.e. a n bit **shift-registry** that stores the **global history** of the program (i.e. the last n branch results). The BHR is basically the same component used to control the multiplexer in a CBP. The BHR is n bytes long.
- A **Prediction History Table** (PHT), i.e. a **table** (2 bits for entry) that stores the **local history** of the program. This table is indexed using the BHR, this means that the global history indexes the local history and that we have 2^n entries in the PHT.

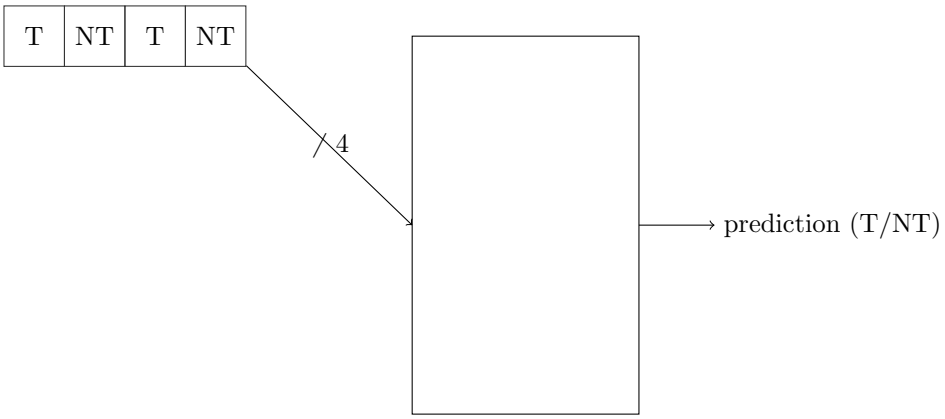


Figure 2.5: A 2-Level Adaptive Branch Predictor.

Notice that a 2LABP doesn't uses branch instruction addresses, thus there is no correlation between the instruction executed and the prediction. In other words, the prediction depends only on the previous prediction (i.e. on the global history). A graphical representation of a 2LABP is shown in Figure 2.5.

2.3.1 G-share

A 2-Level Adaptive Branch Predictor doesn't depend on the instruction address. To add this dependency we can xor the BHR output with the program counter and use the result to index the PHT.

2.4 Branch address buffer

A Branch Address Buffer is a cache storing the Predicted Target Address (PTA), this means that the processor has to query the BAB to obtain the address of the instruction to fetch.

A BAB is made of multiple entries; each entry contains

- The tag of the branch address.
- The Predicted Target Address (PTA).

A BAB in addressed using the index of the branch instruction address. The index A_i can be used to address the BAB and obtain the tag T of the branch address and the PTA. The tag T just obtained is compared with the tag of the instruction A_t and if the two are the same, the PTA is used as branch address; otherwise the next instruction is used.

2.4.1 Complete dynamic branch predictor

If we notice that both a Branch Outcome Predictor (in all of its implementations we have analysed) and a Branch Address Buffer use the index of a branch instruction's address (i.e. the k least-significant bits), we can build a Dynamic Branch Predictor putting the BOP and the BAB side by side. This allows to get, with a single access and using the index,

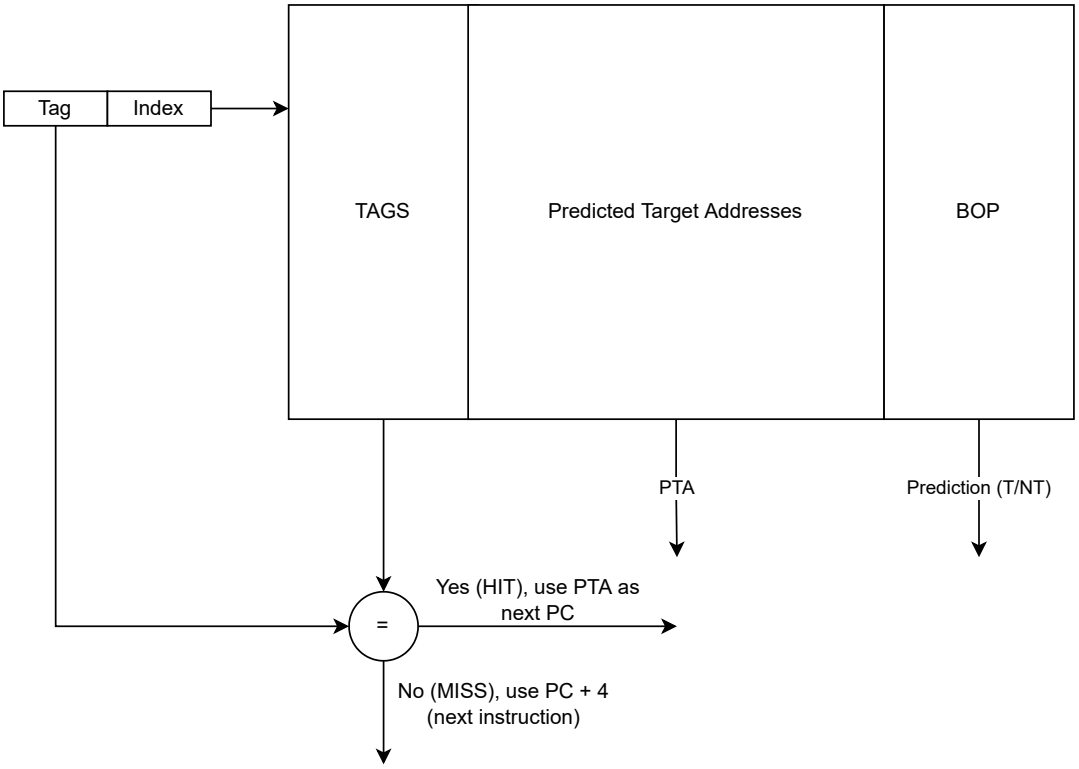


Figure 2.6: A complete dynamic branch predictor.

- The tag in the BAB.
- The PTA in the BAB.
- The prediction in the BOP.

A graphical representation of a complete dynamic branch predictor is shown in Figure 2.6.

Chapter 3

Exception handling

3.1 General definitions

A fundamental part of a processor it the handling of exceptions and interrupts (e.g. I/O interrupts or memory errors). As a reminder, a processor can execute instructions in two different modes

- **User** (U).
- **Supervisor** (S).

and the mode is specified using one or more bits in the Program State Register. The processor has two different stacks, one for each user and when it has to change from a mode to the other (`syscall` and `sysret`) it has to switch stack, too. Every interrupt and exception is associated to a handler (i.e. a set of instructions) that has to be executed in S mode every time an exception (or an interrupt) is called. In a nutshell, when a processor has to call an interrupt handler it

1. Saves the Program Counter.
2. Disables all interrupts if the handler is not interruptable.
3. Swithces to S mode and changes the stack (from uStack to sStack).
4. Sets the Program Counter to the program counter to the handler so that it can start executing the handler.
5. Executes the handler.
6. When the handler is over, it restores the PC (the one it had saved).
7. Switches back to user mode (i.e. the uStack is restored).
8. Continues executing the program it was executing before the interrupt.

From now on, for brevity, we refer to interrupts and exceptions interchangeably.

3.1.1 Classification of exceptions and interrupts

The first classification we are going to mention is between

- **Asynchronous** interrupts. Asynchronous interrupt can happen anywhere during the execution of an instruction and are usually handled between instructions (i.e., after an instruction exits the pipeline). Some examples of asynchronous interrupts are `SIGKILL` from the user or I/O operations. Asynchronous interrupts cause program termination.
- **Synchronous** interrupts. Synchronous interrupts precisely happen during a pipeline stage and are handled within the instruction itself. Synchronous exceptions are triggered by the very instruction that is executing. For instance say the processor is fetching the address of an instruction, if something is wrong with the instruction's address, then the processor has to rise an exception and handle it. In case of synchronous interrupts the instruction must be stopped, the handler is executed, and the instruction is restarted.

Notice that the synchronicity of interrupts refers to the instruction execution (i.e. we say that an interrupt is synchronous with respect to an instruction's execution). Interrupts can also be divided in

- **User requested**. User requested interrupts are requested by a user and are predictable. An example of user requested interrupts in an I/O interrupt. Interrupts of this type can be handled after the execution of an instruction.
- **Coerced**. Coerced interrupts are forced by the system, thus they usually are fully unpredictable.

We can also split interrupts in

- **Maskable**.
- **Non-maskable**.

from the point of view of the user. The difference between the two is whether the processor responds to the exception or not. Finally, another way to classify interrupts is between

- **Resuming** interrupts. Resuming interrupts allow the program to restart after the interrupt has been handled.
- **Terminating** interrupts. Terminating interrupts kill the program after the interrupt has been handled.

3.1.2 Precise interrupt mode

Asynchronous instructions require the processor to run the interrupt handler in **precise interrupt mode**. This means that if the interrupt is handled at instruction i (i.e. we stop execution at instruction i, without executing instruction i), every previous instruction has to be committed or stored (i.e. written in a register or to memory). Basically, we are inserting a break-point between committed and uncommitted instructions. This allows us to

- Precisely define a point from which we can restart execution.
- Treat an interrupt as a misprediction and flush all uncommitted instructions.

Formally,

Definition 1 (Precise interrupt)**.** *An interrupt or exception is precise if there is a single instruction (or interrupt point) for which all instructions before have committed their state and no following instructions (including the interrupting instruction i) have modified any state.*

Practically, when a process wants to handle an instruction it

1. Stops at instruction i.
2. Waits that all previous instructions have been committed.
3. Saves the PC of i.
4. Handles the instruction normally (as we have seen before).

3.2 Handling interrupts

Interrupts model Since the MIPS processor has 5 stages but in the last data is already committed, an interrupt can happen in one of the 4 stages before WB, in particular

- In the IF stage we usually have PC address exceptions or instruction exceptions.
- In the ID stage we usually have instruction's opcode errors.
- In the EX stage we usually have algebraic related errors like division by 0 or overflow errors.
- In the MEM stage we usually have data address exceptions (e.g. we are trying to access a protected zone of the memory).

In addition, an asynchronous exception can rise at whatever moment.

3.2.1 Synchronous interrupts

Let us consider for now synchronous interrupts. Handling single interrupts in trivial; the problem rises when two or more operations generate multiple errors. Before defining a method for handling exceptions, let us analyse how multiple errors can happen and what the desired behaviour is.

The easiest situation to handle is the one in which two operations generate one error each and at the same time. An example is shown in Figure 3.1 (the error is shown as a red square). In this situation we want to handle the exception of the first operation before

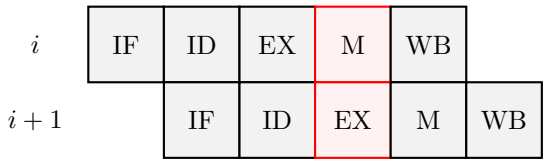


Figure 3.1: Two instructions generate an error at the same time.

Let us consider now a more complex case. Say that a instruction $i+1$ generates an error before an instruction i. An example of this behaviour is shown in Figure 3.2. In this situation the second error is discovered before but we would like to handle the one risen by the first operation before.

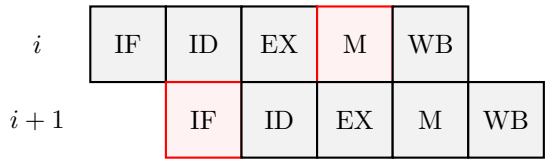


Figure 3.2: A later instruction that generate an error before its predecessor.

Finally we can consider the case in which an operation generates multiple errors (like in Figure 3.3). In this case we want to initially handle the error rose by instruction i and then handle, in order, the errors risen by instruction $i+1$.

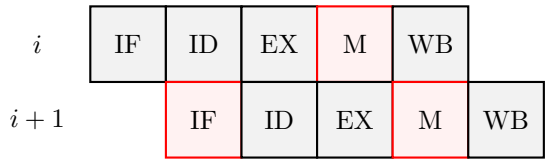


Figure 3.3: A later instruction that generate multiple errors before its predecessor.

Now that we have an idea over what can happen, we can try to find a solution to this mess. In particular, we can remember that after the MEM stage, an instruction is committed, thus we can think at the MEM stage as a synchronisation point where we can decide what interrupt to handle. This means that when an exception rises, we have to delay it until the MEM stage. To achieve this goal we need to extend the MIPS processor and add some new intermediate registers after each stage that store and propagate the exceptions through the pipeline (a register after stage S stores the exceptions rose at the stage before). Exceptions are propagated until they reach the MEM stage where the processor can decide which exception has to be handled.

Notice that this circuit is also able to handle asynchronous interrupt, in fact they can be propagated to the MEM stage, too. Also remember that the circuit has to propage the program counter so that, when we restart execution, we can restart from the exact point where we stopped (i.e. from the instruction that caused the error).

Stalling When the processor realises that an interrupt has to be handled (i.e. at the end of the MEM stage), it has to flush all operations (that are therefore not committed) and execute the fisrt instruction of the handler. A visual representation of this behaviour is shown in Figure 3.4. As we can see, the processor has to pay 4 clock cycles to execute an handler.

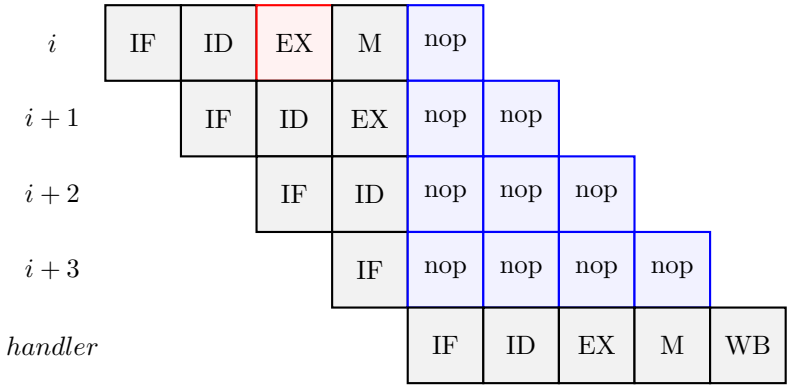


Figure 3.4: Flushing instructions before executing an handler.

Part II

Instruction level parallelism

Chapter 4

Hazards

When executing instructions, a processor has to pay an high cost for stalls that might be caused by data hazards (but not only). In practice, the ideal number of instructions per clock is 1 (i.e. the processor fetches 1 instruction each clock cycle)

$$IPC = 1$$

and therefore the number of clock cycles needed to fetch an instruction is also 1

$$CPI_{ideal} = \frac{1}{IPC} = 1$$

However, what we actually get is that the number of clock cycles increments because of stalls

$$CPI_{pipeline} = CPI_{ideal} + \text{structural stalls} + \text{data hazard stalls} + \text{control stalls} + \text{memory stalls}$$

Notice that if the CPI increases, the number of instructions per clock cycle decreases, thus the processor performs worst.

$$IPC_{pipeline} = \frac{1}{CPI_{pipeline}}$$

That being said, we have understood that a vital part of increasing processing speed is reducing hazards. Before giving a look at some techniques to achieve this result, it's better to categorise hazards, in particular there might happen

- **Structural hazards**. Structural hazards are generated from the fact that the processor needs more hardware resources.
- **Data hazards**. Data hazards happen when two instructions work on the same data that hasn't been committed yet (e.g. an instruction reads a datum not written by the previous instruction). Such hazards can be solved using forwarding (e.g. EX/EX, MEM/EX) or scheduling (statically by the compiler).
- **Control hazards**. Control hazards are generated by branch mispredictions.

Notice that increasing the length of the pipeline increases the impact of hazards on performance because a stall is propagated, on average, on more stages. Another thing to keep in mind is that, by increasing the length of the pipeline, we improve throughput (more instruction are executed in parallel) but not latency because an instruction is committed after more stages.

4.1 Dependencies

Recognising if two instruction are dependent one from the other is fundamental to achieve parallelism, in fact two dependent instructions can't be executed in parallel (but only in order or partially overlapped). Dependencies can be divided in three categories

- **Data dependencies**.
- **Name dependencies**.
- **Control dependencies**.

Let us analyse such dependencies more in detail.

4.1.1 Data dependencies and hazards

A data dependency, can potentially generate a data hazard (that forces the pipeline to stall). There exist three types of data hazards

- **Read After Writes** (RAW). A read after write dependency is generated by an instruction `Ij` that reads a value of a register written by a previous instruction `Ii`.

```
Ii: r3 <- r1 op r2
Ij: r4 <- r3 op r5
```

- **Write After Reads** (WAR). A write after read dependency is generated by an instruction `Ij` that writes a value on a register that has to be read by a previous instruction `Ii`.

```
Ii: r3 <- r1 op r2
Ij: r1 <- r4 op r5
```

- **Write After Writes** (WAW). A write after write dependency is generated by an instruction `Ij` that writes a value to a register also written by a previous instruction `Ii`.

```
Ii: r3 <- r1 op r2
Ij: r3 <- r4 op r5
```

Notice that dependencies are hazards are different, in fact

- Dependencies are **properties of the program**.
- Hazards are **properties of the pipeline**.

Furthermore a dependency doesn't necessarily generate an hazard.

4.1.2 Name dependencies

A name dependency occurs when two instructions use the same register or memory location (called name), but there is no flow of data between the instructions associated to that name. In other words, in name dependencies there is no value that is transmitted between the two instructions but they only use the same register or name. Name dependencies can be further divided into

- **Anti-dependencies**. An anti-dependence occurs when an instruction I_j writes a register or memory location that instruction I_i ($i < j$) reads.

  ```
  Ii: r3 <- r1 op r2
  Ij: r1 <- r4 op r5
  ```

  To prevent unwanted behaviours, the original instruction ordering has to be preserved to ensure that I_i reads the correct value. Notice that anti-dependencies can generate write after read hazards (because we write a register after reading it).

- **Output dependencies**. An output dependence occurs when two instructions I_i and I_j (with $i < j$) write the same register or memory location.

  ```
  Ii: r3 <- r1 op r2
  Ij: r3 <- r4 op r5
  ```

  The original instruction ordering has to be preserved to ensure that I_i reads the correct value, to ensure that the final value of `r3` is the one written by I_j. Notice that output dependencies can generate write after write hazards.

Register renaming

Name dependencies are not data dependencies because there is no data-flow between the instructions, thus they can be avoided changing the register or the memory location used by one of the two instructions. For instance, the instructions in the two examples seen before could be rewritten, using register renaming, as

```
Ii: r3 <- r1 op r2
Ij: r4 <- r4 op r5
```

and

```
Ii: r3 <- r1 op r2
Ij: r4 <- r4 op r5
```

As we can see, we neither write on the same register read nor we write on the same register twice. This means that the instructions I_i and I_j aren't dependent anymore and can be executed in parallel without compromising the behaviour of the program. This solution can be applied both statically (i.e. by the compiler) or dynamically and works well when the processor has many registers.

Memory disambiguation

Register renaming refers to data dependencies in which registers are involved. The same solution can be applied to memory location, but it's a bit more difficult. In particular we have to solve the memory disambiguation problem, in fact two addresses may refer to the same location even being different. Consider for instance the instructions

```
store $1, [$2+2]
load  $3, [$4+3]
```

where the notation `[$2 + 2]` represents the memory pointed by the address in `$2` plus 2 (e.g. if `$2 = 0x1002`, then `[$2 + 2]` points to `0x1004`). In this case, the processor doesn't know, a priori, if `[$2 + 2]` and `[$4 + 3]` point to the same memory location (e.g. `$2 = 0x1001` and `$4 = 0x1000`).

4.1.3 Control dependencies

A control dependence determines the ordering of instructions and is preserved by two properties

- Instructions has to be executed in program order to ensure that an instruction that occurs before a branch is executed before the branch.
- Detection of control hazards has to ensure that an instruction (that is control-dependent on a branch) is not executed until the branch direction is known.

Notice that control dependencies are not critical properties to be preserved.

4.1.4 Program correctness

The most important thing to keep in mind when applying instruction level parallelism is that the program under execution has to be correct (i.e. has to give the same result as if instruction were executed one after the other). In particular we have to enforce two properties to ensure a program's correctness

- **Data flow**. The actual flow of data values among instructions that produces the correct results and consumes them has to be preserved.
- **Exception behaviour**. Any changes in the ordering of instruction execution must not change how exceptions are raised in the program.

Chapter 5

Multi-cycle pipeline

A multi-cycle pipeline allows to execute floating point operations (additions, multiplications and divisions) in a single clock cycle. To achieve this goal we have to add some hardware, and some stages, that allow us to execute floating point operations.

5.1 Complex multi-cycle in-order pipeline

The first type of multi-cycle pipeline that we are going to analyse can fetch one instruction per clock cycle (at best) and issues instructions in order. This type of pipeline introduces some new components and stages, in particular the new pipeline is made of

- An **IF** stage that is identical to the IF stage of a normal MIPS pipeline.
- An **ID** stage that contains two register files, one for integers and one for floating point values. Notice that the size of the latter is doubled with respect to the former, in fact a floating point register needs 64 bits. Remember that the ID stage decodes the instruction, gets the values needed from the registers and stores the values of the WB stage in the same clock cycle.
- Three **EX** stages. These stages contain three more ALUs that are able to compute floating point operations and span multiple stages. In particular these stages contain
  - The integer ALU that needs one stage (EX1). The next state (E2) replaces the MEM stage while the last stage (EX3) is empty.
  - The floating point add ALU that needs three stages (EX1, EX2 and EX3) to compute the sum of two floating point numbers.
  - The floating point multiplication ALU that needs three stages (EX1, EX2 and EX3) to compute the product of two floating point numbers.
  - The floating point division ALU that needs three stages (EX1, EX2 and EX3) to compute the division of two floating point numbers (not sure about this).
- A **WB** stage that behaves like in a normal MIPS pipeline.

The commit point of an instruction is placed at the end of the three EX stages. This means that in some cases (e.g. for integer operations and memory loads or stores) we are wasting one clock cycle (because an `add` needs only 5 stages, while we are using 6) but we are ensuring that all instructions

commit at the same time. Better said, we ensure **in-order commit** (and issue) **of instructions**. This property

- Preserves precise exception model.
- Avoids the generation of WAR and WAW hazards.

Misses Notice that the memory stage might require multiple cycles access time due to instruction and data cache misses.

5.2 Complex multi-cycle out-of-order pipeline

Multi-cycle in-order pipelines work fine but we can also try to drop one assumption and allow the processor to commit instructions out of order. This means that we are going to remove the single commit point and, when an instruction I stalls, the other instructions can keep executing and commit before I. Given the new assumptions we can build a new pipeline made of

- One IF stage that fetches one instruction per clock cycle.
- One ID stage that decodes the instruction. Notice that, differently from the in-order pipeline, the ID stage doesn't get the values from the registers.
- One IS (Issue) stage that get the registers' values needed by the instructions. As for in-order pipelines, this stage contains both 32-bit integer registers and 64-bit floating point registers.
- Multiple execution stages in parallel, in particular
  - An EX stage for integer operations followed by the MEM stage (as for basic MIPS pipelines). Notice that the MEM stage can be bypassed by integers operations and the ALU result can be sent directly to the WB stage. The MEM stages is used instead by loads and stores.
  - Three EX stages for floating point additions.
  - Three EX stages for floating point multiplications.
  - Three EX stages for floating point divisions.
- One **WB** stage that behaves like in a normal MIPS pipeline (i.e. writes values in the register files).

A schematic representation of a complex multi-cycle out-of-order pipeline is shown in Figure 5.2.

As for in-order pipelines, this architecture may need multiple cycles to access memory due to cache misses (which are unpredictable at compile-time) but, in addition, it can also generate WAR and WAW hazards. As always such hazards can be solved using stalls or renaming (as we have seen, WAR and WAW are generated by naming dependencies).

An example of valid execution of (part of) a program is shown in Figure 5.1. Notice that in the example the last instruction has generated a memory error.

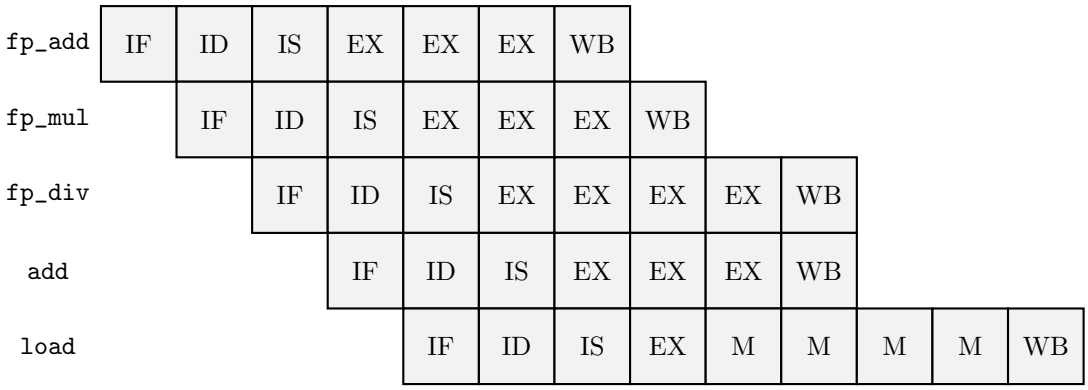


Figure 5.1: A valid execution of part of a program by a multi-cycle out-of-order pipeline.

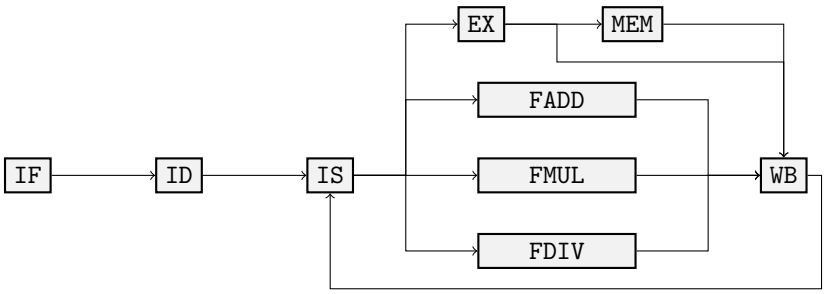


Figure 5.2: A schematic representation of a complex multi-cycle out-of-order pipeline.

Chapter 6

Scheduling and multi-issue pipelines

Instruction level parallelism can be used to increase the number of instructions fetched per clock cycle (i.e. $IPC > 1$) and therefore decrease the clock cycles per instruction (i.e. $CPI = \frac{1}{IPC} < 1$). To achieve this goal we have to contemporary issue independent instructions in the IF stage without generating hazards. This means that it's fundamental to check if two instructions are dependent if we want to execute them in parallel, in fact we want to avoid any type of hazard. Basically, we are trying to overlap the execution unrelated instructions.

6.1 Multi-issue pipeline

The first step in obtaining parallelism is building a pipeline that can issue more instructions per clock cycle. For now, we are going to analyse two instructions parallelism. This means that we will describe a pipeline that, ideally, can issue two instructions per clock cycle (i.e. $IPC = 2$ and $CPI = 0.5$) or equivalently, that the new pipeline will have double throughput (but same latency). For simplicity we will consider the basic 5 stages MIPS pipeline but the same ideas can be applied to more complex pipelines. Remember that, **two instructions can be issued in parallel only if they are independent** one to the other, then the ideal CPI can't always be reached.

Pipeline structure Since we are adding some functionality, we have to modify the pipeline a little, in particular

- The **IF** stage is able to fetch two instructions. Such instructions can't be two operations, nor two memory instructions (e.g. the processor can't fetch `add, mul` or `load, store` or `add, and`).
- The **ID** stage decodes the instructions and retrieves the values needed from the register file. It's important to underline that since we are executing a single process, both instructions operate on the same register file (i.e. we have only one register file). Also notice that the register file has to be modified in order to have 4 read ports (2 for each instruction) and 2 write ports (1 for each instruction, used by the WB stage).

- The **EX** stage needs two ALUs, otherwise it wouldn't be able to execute two different computations in one clock cycle. Let us highlight the fact that one ALU should be used for arithmetic instructions while the other for memory instructions (i.e. compute the memory address). This means that we can execute in parallel only arithmetic and memory instructions. However, the pipeline can be further modified to allow the execution of multiple arithmetic instructions.
- The **MEM** stage is unchanged with respect to the basic pipeline, in fact only one of the two parallel instructions can write to or read from memory.
- The **WB** stage allows to write the result of both instructions to the register file (which is in the ID stage).

Issue-width problem The issue-width is the number of instructions that can be issued in parallel in a clock cycles. This number is limited because, if we would pick a big number it would be too hard to find instructions that can be issued in parallel, hence the performance wouldn't improve too much.

6.2 Scheduling

To allow multiple instructions to be executed in parallel without compromising the execution of the program we have to change the order in which instructions are issued remembering that dependent instructions can't be executed in parallel. Scheduling (i.e. ordering instructions) can be done

- **Statically**. Static scheduling allows the compiler to detect instructions' dependencies and order them in order to obtain maximum parallelism.
- **Dynamically**. Dynamic scheduling allows the processor to locate parallel instructions and executing them. As for now, this solution is dominating the processors' market.

6.2.1 Dynamic scheduling

Dynamic scheduling allows the processor to execute an instruction I_i, independent from I_j, even if the latter has stalled. This solution improves throughput because otherwise I_i should have waited for the stall to end, thus losing clock cycles. Notice that instructions are fetched and issued in program order (in-order-issue) but this technique can generate out-of-order commits, hence WAW and WAR hazards can arise and we can have unordered exceptions. The first problem has to be solved (using renaming) but the second can be neglected, in fact we might want to allow unordered exceptions.

To better understand how dynamic scheduling works, let us consider the following piece of code

```
divd f0, f2, f4
addd f10, f0, f8
subd f12, f8, f14
```

executed on a multi-issue processor. As we can see, there is a dependency between the first two instructions that may cause a read after write hazard. However, the third instruction is independent from the previous, thus it can be executed in parallel with one of the previous two (if we forget for a moment that only arithmetic and memory instructions can be executed in parallel). Basically the result we get is shown in Figure 6.2.

Advantages and disadvantages The main advantage of dynamic scheduling is that we can obtain a **CPI smaller than 1**. This comes however at a cost, in fact

- Processors are **very expensive in terms of area and power consumption** because of their complex control logic.
- Processors are **not scalable**.
- The **cycle time is limited** by the scheduling logic.
- The **design verification of the processors can be extremely complex** because of the complexity of the scheduling logic.

For these reasons, dynamic scheduling is usually used on commercial, general purpose processors that have no power-related issue.

Superscalar processors

Determining if two instructions can be executed in parallel is a very intensive task, hence processor have dedicated circuitry for this purpose. A typical examples of processors that can execute dynamic scheduling are superscalar processors. A superscalar processor leverages all dynamic optimisations to reach the best IPC, in particular it features

1. **Pipelining**.
2. **Dynamic-scheduling** (hence out-of-order execution).
3. **Multiple-issue** (in-order).

Architecture Superscalar processors are multiple-issue processors with three stages

- The first stage contains the **instruction fetch** and **decode** unit. This unit can **dynamically** handle multiple instruction fetch and decoding per cycle. In this stage, the hardware also decides (dynamically) which instructions can go in parallel and issues them. Remember that in superscalar processors, **instructions are issued in order**.
- The execution stage contains many functional units, usually two integer ALUs, one floating point ALU, one branch unit and a load/store unit. Every unit handles different operations and needs a different number of clock cycles. This means that the processor can **execute out-of-order**.
- The commit unit can commit in-order or out-of-order and has to check if there are any dependencies between instructions. To solve dependencies, the commit unit can
  - Use an **in-order buffer** that stores the instruction and commits them only when all previous instructions have arrived (i.e. in order). The main disadvantage of this method is that we increase latency.
  - Use **register renaming**. Register renaming can be done either statically (by the compiler) or in the first stage. Register renaming allows the processor to commit out-of-order.

The general representation of a superscalar processor is shown in Figure 6.1.

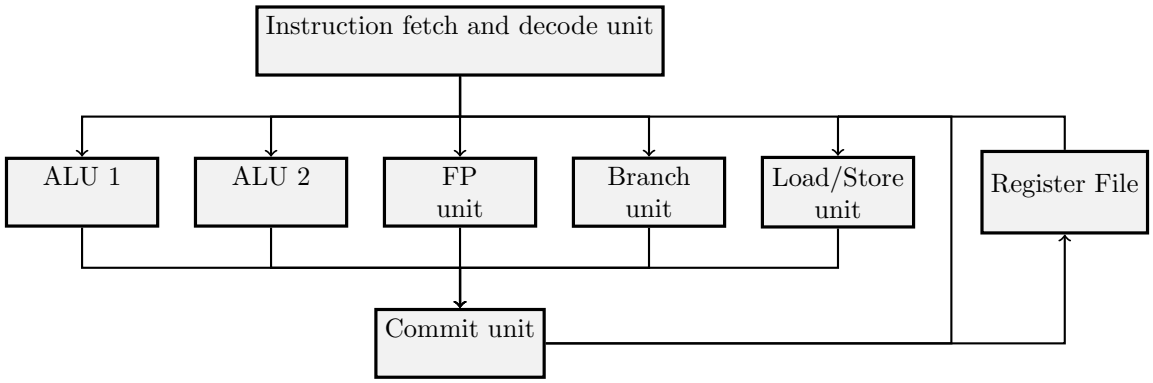


Figure 6.1: The structure of a superscalar clock.

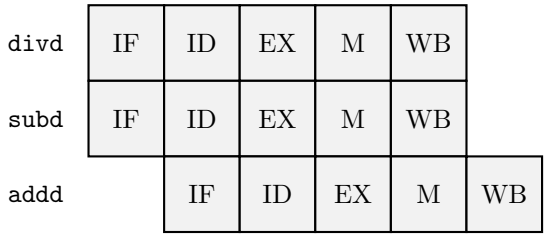


Figure 6.2: Dynamic scheduling of an independent instruction.

6.2.2 Static scheduling

Static scheduling allows the compiler to look for independent instructions and maximise parallelism. This means that the processor receives blocks of instructions that can be executed in parallel. One might ask what happens if the compiler can't find a couple of instructions that can be executed in parallel. In such cases, the compiler inserts a `nop` (notice that if a `nop` is executed we aren't losing in performance with respect to a basic pipeline because we are executing a `nop` instruction and an effective instruction). Also remember that, compilers can only search independent instruction in basic blocks, otherwise we should be handling branches in parallel. This can be a problem for performance because usually 15% to 25% of operations are branches and in general a basic block is made of 7 instructions, hence the compiler has few instructions to look for independent ones and the parallelisation power can be limited. To actually increase the throughput we have to find ways to allow the compiler to search instructions in different basic blocks.

Disadvantages Static scheduling is very powerful but it also has some drawbacks, in particular it has to deal with

- **Unpredictable branches**.
- **Variable memory latency** (hence unpredictable cache misses).
- **Code size explosion**.
- **Compiler complexity**.

- **Code portability**.
- **Performance portability**.

Very Long Instruction Words processors

Very Long Instruction Word (VLIW) processors expect dependency-free code generated by the compiler. This means that the compiler (and the compiler only) handles all dependencies statically. When multiple instructions can't be handled together (i.e. no parallelism can be achieved) a `nop` operation is issued. Notice that the fact that the compiler handles all dependencies doesn't mean that no stalls are inserted. In particular, cache misses and branch mispredictions can't be predicted by compilers, thus in such cases the processor has to insert stalls. Since parallelism comes from the compiler, the processor only has to assign each operation to the correct functional unit.

Another thing that has to be kept in mind is that the compiler has to know the latency of each work unit (i.e. the architecture of the processor) to schedule instructions correctly. In particular, if a functional unit is still in use, the compiler can't issue another instruction that uses the same functional unit, otherwise we would have a structural hazard in the first stage. Compilers can also solve WAW and WAR dependencies using renaming.

One final thing to underline is that since the code may contain many dependencies, the compiler will issue many `nop`s and the instruction cache will contain many `nop`s. This means that we need to implement some sort of compression to save some space.

Advantages and disadvantages The main advantage of VLIW processors is that **they are much simpler than superscalar processors, thus occupy less area and consume less power**. However,

- The compiler has to be optimised to get high performances.
- The code compiled for a device or an architecture can't be ported to another architecture or device because the code has been optimised and the compiler has to know the latency of the processor.
- The number of instructions that can be executed in parallel is very limited because the compiler has to search in basic blocks that contain only few instructions that can depend on to the other. To solve this problem, modern computers use
  - Multi-core.
  - Multi-threading.
  - Additional processors like vector processors and GPUs.

For these reasons, VLIW is usually used on mobile devices because the processor has to be power efficient while the code only needs to run on a specific device.

Chapter 7

Dynamic scheduling

7.1 Scoreboard

The scoreboard is an architecture for dynamic scheduling, single in-order issue processors with no forwarding. A processor is made of

- An **issue** stage.
- A **read** stage.
- Four **functional units**.
- A **register file**.
- A **scoreboard**.

In any of the stages above detects an hazard, execution is stalled until the hazard isn't solved.

7.1.1 Issue stage

In the issue stage, the processor

- Decodes the instruction.
- Checks for structural hazards. In particular the processor has to check if the functional unit needed by the instruction is free.
- Checks for write after write hazards, and stalls if some are present.

An important thing to remember about the issue stage is that instructions are issued **in-order**. This means that if instruction `i+1` is stalled, we can't issue instruction `i+2`, even if it's independent from `i+1` and could be executed. Furthermore, an instruction is issued only if it's not dependent on the instructions currently executing.

7.1.2 Read stage

In the read stage, the processor

- Checks for read after write hazards.
- Checks for structural hazards. In particular, the processor has to check if the number of read ports of the register file is enough to get all the values requested by the instructions in this stage. Notice that the scoreboard architecture can read two instructions per cycle, thus the register file has 4 read ports. However, multiple instructions might be waiting (because stalled) in this stage, thus the number of ports might not be enough (e.g. a stalled instruction and 2 issued instructions).

The read stage is the first one in which we can get out-of-order execution, in particular some operands might be ready before others.

7.1.3 Execution stage

The scoreboard architecture can use four functional units with different latency, thus the processor can complete the execution out-of-order because

- The read stage can go out-of-order.
- Instructions on different functional units need a different time to complete.

In particular the four functional units handle

- **Floating point division**.
- **Floating point multiplication**.
- **Floating point addition**.
- **Integer addition**.

The execution stage (in particular the integer functional unit) handles memory interaction, thus cache misses can occur and latency can increase.

7.1.4 Write stage

The write stage handles writes to the register file. Since the execution stage can go out-of-order, in this stage we can have out-of-order commits. In this stage, the processor

- Checks for **write after read hazards**. In particular, before writing a value on a register we have to check that all previous instructions have read the value in the register file.
- Check for **structural hazards**. In particular, multiple instructions might want to read the register file but it only has one write port.
- Check for **write after write** hazards. Notice that in the basic version of the scoreboard, this check is done in the issue stage. However, if we can delay the check we can start issuing and executing some instructions without stalling in the issue stage hoping that the hazards solves during execution (because of out-of-order execution).

As always hazards have to be solved with stalls.

7.1.5 Scoreboard

The scoreboard is a table that decides

- When an instruction can read its operand and begin execution. If the scoreboard decides the instruction cannot execute immediately, it monitors every change and decides when the instruction can execute.
- When an instruction can write its result into destination register.

To take such decisions, the scoreboard stores

- The **instruction status**. The instruction status structure stores, for each instruction, the stage in which it is.
- The **functional units status**. The functional units status structure stores, for each unit
  - If the unit is busy.
  - What instruction (i.e. the op code) is in execution (if any) in that unit.
  - The destination register F_i of the instruction.
  - The source registers F_j and F_k of the instruction.
  - The functional units Q_j and Q_k producing registers F_j and F_k.
  - Two flags that say if the registers F_j and F_k are ready. After the operands are read, these flags are set to false.
- The **register result**. The register result structure stores which functional unit will write each register (blank if no pending instructions will write a register).

7.2 Tomasulo Algorithm

The Tomasulo architecture is an in-order issue, out-of-order completion architecture that allows a processor to

- Process independent instructions behind a stall (i.e. **dynamic scheduling**).
- Avoid write after read and write after write hazards using **implicit renaming register renaming**.

Before defining the architecture, let us analyse how register renaming can solve WAR and WAW conflict. Let us, for instance, consider the instructions below

```
add F6, F0, F8
store F6, 0(R1)
mul F6, F10, F8
```

As we can see, there are multiple conflicts

- A read after write on `F6` between instructions 1 and 2.
- A write after read on `F6` between instructions 2 and 3.
- A write after write on `F6` between instructions 1 and 3.

If we rename register `F6` to `S` in the first two instructions we can solve the WAW and WAR conflicts

```
add S, F0, F8
store S, 0(R1)
mul F6, F10, F8
```

because

- The `mul` instruction doesn't write the same register as the `add` instruction.
- The `mul` instruction doesn't write the same register read by the `store` instruction.

Notice however that the read after write conflict is still present.

7.2.1 Reservation stations and store buffers

Register renaming is implicitly implemented (i.e. a register isn't actually renamed, but we obtain the same result) using

- **Reservation stations** for arithmetic and logic operations.
- **Store and load buffers** for memory operations.

A reservation station (RS) is a buffer that temporary store the result of an operation or of a register. Multiple reservation stations are placed before each functional unit of the processor, thus we have multiple reservation stations for each type of instruction. This also means that, differently from the scoreboard architecture, the Tomasulo architecture implements a distributed control system. Store and load buffers works similarly to reservation stations but are placed in front of the memory unit, in particular these buffers store the pending memory instructions that are waiting to us the memory unit.

RAW conflict and forwarding

The result of a functional or memory unit computation is stored in the reservation stations using a Common Data Bus, hence we can do one write per clock cycle. This however is useful because it implements some sort of forwarding that can be used to solve read after write conflicts.

Structure of a reservation station

Each reservation station contains

- A **tag** that uniquely identifies the station.
- A **busy flag** that tells if a value is stored in the station.
- A **OP** field that stores what type of operation is the station used for. Notice that even if a RS is placed in front of a specific unit, different operations can be executed (e.g. an add functional unit can execute `add`s, `addi`s, `sub`s, ...).
- The **values** V_i, V_j of the **source operands** of an instruction. In case of store buffers, we only save one operand V_j which is the memory address for the load or store operation.

- The **pointers Q_i, Q_j to the reservation stations** (i.e. to their tags) that produce V_i and V_j. These pointers are needed because some values might not be ready yet, thus we can only store where the source values will be, but not the actual values. If these pointers have value 0, then the source operands are already available in V_i and V_j. Also remember that, for each operand, only one between the pointer and the value is valid.
- Store buffers also save the **reservation station in which the result to store is**.

If the reservation station is a

How reservation stations are used

When an instruction is issued, the operands' registers are replaced by their values or by a pointer to a reservation station. Notice that the Tomasulo algorithm implements dynamic scheduling because every decision about renaming is done by the hardware at runtime. Moreover, the number of reservation stations is higher than the number of registers, thus the processor can optimise more than the compiler.

7.2.2 Pipeline stages

The Tomasulo pipeline is divided in three stages

- The **issue** stage.
- The **execution** stage.
- The **write result** stage.

Issue stage

In the issue stage the processor picks the instruction `i` on the head of the instruction queue (in-order issuing) and checks

- If the reservation stations or the store buffer for the issuing instruction are free. If not we have a structural hazard that can only be solved with stalls.
- If there are any write after write (WAW) or write after read (WAR) hazards. In particular
  - For WAR conflicts, if instruction `i` writes `Rx`, read by an instruction `k` already issued, `k` knows already the value of `Rx` read in the reservation station or knows what instruction (previously issued) will write it. So the register file in the instruction can be linked to `i`.
  - For WAW conflicts, since we use in-order issue, the register file can be linked to `i`.

Execution stage

In the execution stage the processor starts execution only if

- No read after write conflict is present. Practically, the processor has to wait that all operands in the reservation stage are ready (i.e. pointer to 0). If at least one operand is not ready, the functional unit should wait (i.e. stall) for the Common Data Bus to write the result on the reservation station. Notice that waiting for the results from the CDB and not from the register file is a sort of forwarding.

- No structural hazard is present. Practically, the execution unit has to be free.

In this stage we also have to assume that no instruction is issued while a branch is pending. If branch prediction is used, CPU must know the prediction's correctness before executing the following instructions. Also notice that, the functional unit is reserved before checking the presence of conflicts. This means that, if we have a single functional unit of a specific type (e.g. a single ALU), all instructions that have to executed on that FU, are executed in order.

Load and store instructions Load and store instructions are divided in two stages

1. The processor computes the effective address when the base register is available and places it in the load or store buffer.
2. The processor
   - executes loads in the load buffers as soon as memory unit is available.
   - waits for the value to be stored in the store buffer before sending it to the memory unit.

   This behaviour helps reducing hazards.

Write result stage

In the write result stage,

- Whenever a result of a functional unit is ready, it's written in all reservation stations, store buffers and in the register file using the Common Data Bus.
- Store instructions also write data to the memory unit when memory address and result data are available.
- The processor marks the registration station that was waiting the result as available.

7.2.3 Advantages and disadvantages

The biggest advantage of the Tomasulo architecture is it's speed, however this comes at a cost, in fact the processor is very complex and need more hardware to implement the control logic; this translates in more power consumption. Another disadvantage of this architecture is that a common bus is used to write the results of a computation. This means that the last stage of the pipeline is the bottleneck of this architecture. This issue can be solved using multiple busses, even if it might increase the processor's complexity.

7.3 Register renaming technologies

7.3.1 Loop unrolling

Implicit register renaming (i.e. reservation stations in the Tomasulo algorithm) can be used for loop unrolling. To introduce this technique let us consider the following example

```
LOOP: ld f0, 0, r1
      multd f4, f0, f2
      store f4, 0, r1
      subi r1, r1, #8
      bne r1, LOOP
```

under the following assumptions

- Branch are always taken.
- A multiplication takes 4 clock cycles.
- The load at the first iteration takes 8 clock cycles because of a cache miss.
- The load at the second iteration takes 1 clock cycle (cache hit).
- The processor has two functional units for multiplication.
- The processor has a single load/store unit.

Hazards should be measured considering multiple iterations, in fact we notice that there is a WAW dependency between the loads (the first one takes 8 clock cycles, hence it's in the second load's range) and the multiplications (multiplications takes 4 clock cycles). One might also notice that in the original code there is a dependency between the `subi` and the `bne`, however we wont consider it because we have assumed branch always taken.

Tomasulo

The WAW (but in general also WAR) hazards in the code above can be solved using the Tomasulo architecture, in fact

- The first load is assigned to a reservation station, say RS_1. This station is used during the instructions of the first iteration.
- The second load is assigned to a different reservation station, say RS_2. This means that, after the second load, the pointer of `f0` in the register file points to RS_2, hence all future operations that use `f0` read the correct value, i.e. the value written by the load in the second iteration (and not the one written by the first load and saved in RS_1). Moreover, the old value (i.e. the one used by the first iteration and saved in RS_1) can still be used for the previous instructions because it's written in a different reservation station.

Register renaming

When a processor doesn't support implicit register renaming and the Tomasulo algorithm, we have to use static register renaming. In other words, we let the compiler replace some registers with others to avoid WAW and WAR conflicts. Notice that this is a static technique and it can still lead to stalls (e.g. for cache misses, structural hazards or data hazards). Let us consider the same piece of code used before

```
LOOP: ld f0, 0, r1
      multd f4, f0, f2
      store f4, 0, r1
      subi r1, r1, #8
      bne r1, LOOP
```

and let us assume that the branch is always taken. The idea behind loop unrolling using register renaming is to write down the instructions of n iterations like they were part of a single iteration. Practically, we can unroll two iterations and obtain the following code

```
LOOP: ld f0, 0, 0(r1)
      multd f4, f0, f2
      store f4, 0, 0(r1)
      ld f2, 0, -8(r1)
      multd f8, f2, f2
      store f8, 0, -8(r1)
      subi r1, r1, #16
      bne r1, LOOP
```

The number `n` is called **unrolling factor** and represents the number of iterations that are unrolled. Notice that the load and stores at the first iteration are applied on `r1` while the one at the second iteration are done at `-8(r1)` because the `subi r1 r1 #8` is applied at the end. Moreover, since we are unrolling two loops, we have to subtract 16 from `r1` because we have condensed the two `subi` instructions. Finally notice that in this case, differently from Tomasulo, we have modified the register used in the two iterations. In particular the second iteration replaces register `f0` with register `f2` and register `f4` with register `f8`.

Advantages and disadvantages The advantages of this solution, with respect to Tomasulo, are

- The overhead instructions (in our example, `subi` and `bne`) are payed only once at the end.
- It's easier to implement (even on a very simple architecture), in fact we only need a good compiler.

This technology has however some disadvantages, in particular

- More instructions are issued to the instruction cache.
- Many registers are occupied. In particular the number of registers occupied is called **register pressure**.
- Branch prediction is critical. If the unrolling factor is n then we must be sure that the loop does at least n iterations. If a prediction is wrong we might have some problems that should be handled appropriately.

From this analysis we can understand that the unrolling factor is a trade-off between register pressure and instructions' overhead.

Rescheduling The compiler can further improve performance and reduce the number of stalls using rescheduling. In particular, if we consider our example, we can put all the `load` operations at the beginning so that all data is ready before starting the multiplications, followed by the multiplications and the stores. Notice that we can put all load instruction at the beginning because we used static register renaming to solve naming dependencies. Moreover, putting the load instructions at the beginning of the code reduces the probability of cache misses because, after the first load, the data (which is store contiguously in a page) is already in the cache. Also remember that the scheduler can use the branch delay slot to exploit the instruction after the branch. In a nutshell, the code in our example can be optimised as follows

```
LOOP: ld f0, 0, 0(r1)
      ld f2, 0, -8(r1)
      multd f4, f0, f2
      multd f8, f2, f2
      store f4, 0, 0(r1)
      subi r1, r1, #16
      bne r1, LOOP
      store f8, 0, -8(r1)  # branch delay slot
```

7.3.2 Explicit register renaming

Another technology used to implement register renaming is called explicit register renaming and consists in adding a physical register file in the processor that can't be used by the compiler (i.e. it can't be accessed by the ISA).

The processor maps each register (i.e. each register used by the compiler) to multiple physical registers. This allows the processor to change the physical register depending on the iteration while still using the same normal register. Say for instance register `r1` is mapped to `p1`, `p2` and `p3`; when an instruction in a loop uses `r1`, the processor actually replaces `r1` with one of `p1`, `p2` and `p3` depending on the iteration (e.g. `p1` for the first iteration, `p2` for the second and `p3` for the third). This type of renaming is done dynamically by the processor.

To implement this technology we need

- A **renaming table** that maps an instruction to a physical register.
- A **freelist** to save physical registers that can be used. When a new instruction is issued, a physical register is picked from the freelist. When a physical register isn't used by any live instruction, it's put back in the freelist.

Explicit register renaming in the scoreboard architecture

Explicit register renaming can be used in a scoreboard architecture to avoid naming hazards (i.e., WAWs and WARs). In particular

- In the **issue** stage, the processor
  1. Decodes the instructions.
  2. Checks for structural hazards.
  3. Allocates a new physical register for the result of the instruction.
- In the **read operands** stage, the processor
  1. Waits until all RAW hazards are solved. This means that all true data dependencies are solved in the read operands stage.
  2. Reads the operands.
- In the **execution** stage, the processor executes the instruction as soon as it receives the operands from the previous stages.
- In the **write result** stage, the processor writes the results in the register file.

7.4 Reorder buffers

All dynamic methods we have analysed achieve parallelism only using instructions in the same basic block (i.e. a set of consecutive instructions not separated by a branch). This means that only independent instructions in the same basic block can be executed in parallel. Reorder buffers allow to extend dynamic scheduling behind branches (i.e. to instructions of different basic blocks). To achieve this goal, the processor needs

- **Dynamic branch prediction**.
- **Speculation**. Speculation means predicting what instruction will be executed after a branch and start executing it (but not committing it) without checking if the branch prediction was correct.
- **Dynamic scheduling**.

The main problem we have when executing instructions after a branch is that we don't know if the prediction is correct or not. To solve this issue, we have to speculate on the next instruction to be executed and undo the instruction (i.e. flush the instruction) if the prediction is wrong. This allows us to execute instructions beyond basic blocks and use such instructions to do dynamic scheduling.

Since we have added the concept of speculation, we should be able to mark an instruction as speculative or not, in fact

- A non-speculative instruction can always be committed because we are sure that the processor has to execute it.
- A speculative instruction can't be committed until we are sure that the processor should execute it. If we commit the instruction and then we realise that we shouldn't have executed it, we can't undo the effect of the instruction, hence we have to recognise a speculative instruction to block it at the commit stage.

In particular, each instruction is marked, using a 1 bit flag, as speculative or not.

7.4.1 Reorder buffer

All speculative instructions can be executed (i.e., can pass through the FUs) but can't commit until the processor is sure that the prediction is correct and that the instruction should be actually executed (i.e., committed). This mechanism is implemented using a ReOrder Buffer (ROB). In particular, the reoreder buffer is used to store the result of an instruction that has finished execution but hasn't committed yet.

The ROB is a circular buffer loaded by the issue stage. In particular, the buffer has two pointers, one to the first instruction (i.e. the head) and one to the first empty slot (i.e. the tail). Instructions are always added to the tail and removed (i.e. committed) from the head. When an instruction is issued, an entry is added to the (tail of the) reorder buffer. This means that **instructions are committed in order** (but can still be executed out-of-order), in fact they are issued in order and they are added to the buffer in the same order they are issued. Since the buffer doesn't change the order or instruction (i.e it's simply a FIFO structure), then the instructions are committed in order. Notice that using reorder buffers, we obtain a precise interrupt model because interrupts are accepted at the commit stage (i.e., when they have finished their execution, when they are retiring) and commits are executed in order. This means that the interrupt are also executed in order.

The reorder buffer is located after the functional units and before the register file.

Instruction commit

An instruction in the register buffer commits when all the following conditions are true

- **The instruction has finished its execution**. An instruction that has finished its execution is called **retiring instruction**.
- **The instruction is not marked as speculative**. A speculative instruction, either is marked as non-speculative when the processor confirms the prediction or is flushed when the processor realises that the prediction is wrong.
- **The instruction is on the head of the buffer**. In other words, all previous instructions have been already committed.

7.4.2 Reorder buffers and reservation stations

Reorder buffers are used together with reservation stations, however RSs have a different role with respect to the Tomasulo architecture. In particular, the pointers in the register stations point to the reorder buffer that have the result of the computation and not to the functional units. This means that, the renaming functionality is done by the reorder buffer and the reservation stations are used only for buffering the instructions' operands. Notice that, the structure of each reservation station doesn't change. Also notice that, the ROB replaces the store buffer used for storing the memory address to write to.

The result of an instruction

- Is passed to the reservation stations by the reorder buffer when the instruction ends its execution.
- Is written to the register file or to memory when the instruction commits.

Structure

Each entry of the ROB contains

- A **busy flag**. The busy flag tells if the entry contains data.
- The **instruction** to which the entry refers to.
- The **status** of entry. An entry can be issued (if it has been issued by it hasn't executed yet), in execution, retiring (i.e., in the execution complete state), or ready to commit. Remember that only an instruction is ready to commit (while many may be in the complete state) and it has to be on the head of the buffer, non-speculative and it has to have finished its execution.
- The **destination** register. The destination register is the register in which the result of the instruction is written.
- The **value field**. The value field is used to hold value of the result until the instruction commits.
- A **speculative flag**. The speculative flag specifies if the instruction is a speculative instruction, i.e., it comes after a branch whose prediction hasn't been checked yet.
- A **head flag** that says if the entry is the head of the reorder buffer.
- A **tail flag** that says if the entry is the tail of the reorder buffer.

7.4.3 Tomasulo architecture with reorder buffers (speculative tomasulo)

Reorder buffers can be used with the Tomasulo architecture to achieve implicit register renaming and loop unrolling beyond basic blocks. The execution of an instruction is divided in 4 phases

- **Issue** (or dispatch).
- **Execution start**.
- **Execution completion**.
- **Commit**.

The main difference with the normal Tomasulo architecture is that pointers are directed towards the ROB entries (which hold the result of an instruction) instead of reservation stations, hence implicit register renaming is achieved thanks to ROB entries.

Issue

Instructions are issued in order and during this phase the processor

- The processor gets instruction from instruction queue.
- Reserves one reservation station and one entry in the reorder buffer. If one of the two or both are not available the processor has to stall (structural hazard).
- If the operands of an instruction are available (i.e. if no RAW hazard is present), they are sent to the reservation station and the number of the ROB entry allocated for result is sent to RSs (to tag the result when it will be placed on the Common Data Bus).

Execution start

The execution stage starts only when all operands in the reservation stations are ready, otherwise it stalls. Notice that, for a store instruction, the processor only has to check if the the value of the base register is available (i.e. the source operand can be available later on). During this stage, the processor also looks for RAW hazards. As always, **execution can go out-of-order**.

Execution complete

When the execution of an instruction is completed, the processor

- Writes the result on the common data bus. In this case the CDB writes the result **only on the reservation stations and not on store buffer and register file**.
- Writes the result on the reorder buffer.
- Marks the reservation station as free.
- For a store instruction, the value to be stored is written in the ROB value field; otherwise the processor monitors the CDB until value is broadcasted.

Commit

In the commit phase, the processor

- Updates the register file (for arithmetic and logic instructions) or the memory (for store instructions) with the result at the head of the reorder buffer.
- Removes the instruction on the head of the buffer (i.e., the one committed).
- Flushes all instructions marked as speculative, if the prediction is wrong.

In other words three sequences of operations can happen,

- A result is written to the register file and it's removed from the ROB.
- A result is stored to memory and it's removed from the ROB.
- All the results of speculative instructions are flushed if the prediction is wrong.

Also remember that **commit are executed in-order** because of the reorder buffer.

Chapter 8

Very Long Instruction Word

8.1 Introduction

A Very Long Instruction Word processor is a **static multi-issue** processor. This means that, multiple independent instructions can be issued at the same clock cycle, resulting in an ideal increase of the IPC (hence a decrease of the CPI). In particular, for a N-issue VLIW processor, the ideal IPC is N while the ideal CPI is $\frac{1}{N}$. However, in the real case, the actual IPC is lower because of stalls and `nops` introduced due to hazards in the code.

The basic idea of VLIW is that, the compiler can decide what instructions can be executed in parallel without generating data hazards (true or naming hazards) and issues them directly in parallel. This means that the CPU only has to handle branch mispredictions and cache misses.

8.1.1 Processor

VLIW CPUs receive very long instructions called **bundle**, that are made of multiple basic instructions called **operations**. A representation of a bundle is shown in Figure 8.1. The bundle is forged

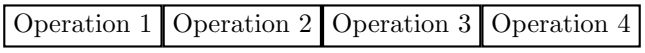


Figure 8.1: A 4-operations bundle.

statically by the compiler that can look into basic blocks of instructions to find some that do not depend one from the other. Each bundle is made of a fixed number of operations each of which is assigned to a different functional unit. The mapping between one operation slot and the FU is fixed, hence the compiler can only insert some type of instructions in each slot.

Nops

Many times, it's not possible for the processor to find an instruction to issue in a slot that doesn't depend (i.e. that doesn't generates hazards) on the others already added to the bundle, so it has to put a `nop` instead. Since the number of independent instructions in a basic block is usually limited, the number of `nop`s inserted is usually quite big, hence the code generated by the compiler is quite large and is full of `nop`s.

To solve this problem, we can have multiple functional units for each bundle slot so that we can solve at least structural hazards. The functional units are controlled by a dispatcher that decides for each slot, on which functional unit of that type the instruction should be executed. For instance, if the first slot is used for load and store operations and it's associated to two load/store functional units, the dispatcher decides to which of the two an operation should be assigned.

8.1.2 Stages

VLIWs can run on the standard MIPS pipeline, however in some cases we can use a modified pipeline with 6 stages,

- The **Instruction Fetch (IF)** stage.
- The **Instruction Decode (ID)** stage.
- The **Register Read (RR)** stage. In the read register stage, the processor reads from a single register file the operands needed by each operation. Since the operations have to read the registers in parallel, the register file of an N-issue processor has to have $2N$ read ports and N write ports.
- The **Execution (EX)** stage.
- The **Write Back (WB)** stage.

8.1.3 Instruction execution

Every functional unit has a different latency, hence execution can go out-of-order and WAR and WAW hazards can be generated. To solve this problem, the compiler has to be aware of the latency and the number of functional units so that it can compute after how many operations the hazards is solved. Moreover, a VLIW processor can exploit fully pipelined functional units (and not only a pipeline architecture). In particular, a single functional unit can execute multiple instructions in a pipeline. The concept of pipeline is the same as for processors, but it's applied to a single functional unit.

8.1.4 Hazards, mispredictions and misses

A CPU has to handle hazards, branch mispredictions and misses. Let us understand how a VLIW processor behaves in each of this situations,

- **Hazards** are avoided statically by the compiler that put `nop`s in the code to replace instructions that can't be execute because of a dependency with a previous instruction. Structural, true data and naming hazards are all solved statically. In particular
  - WAR dependencies can be solved using `nop`s and forwarding (only between operations in different bundles).
  - WAW and WAR dependencies can be solved using `nop`s, forwarding and static register renaming.
- **Branch mispredictions** can't be predicted statically, hence they can't be controlled by the compiler. This means that the CPU has to flush the pipeline when a branch prediction is wrong.

- **Cache misses** can't be predicted statically, hence the only way to solve them is to dynamically use stalls.

8.1.5 Advantages and disadvantages

The main advantages of VLIW processors are

- The compiler has a **longer view on the instructions** that can be scheduled because dynamic schedulers can only look into the instructions in the instruction buffer, while compiler can look into the whole program.
- The **CPU is simpler** with respect to dynamically-scheduled processor, hence less area is occupied and less power is consumed.

The disadvantages of VLIW processors are

- Cache misses and branch mispredictions can't be predicted at compile (i.e., static) time.
- We can only look at basic blocks of code.
- The compiler has to be optimised to get high performances.
- The code compiled for a device or an architecture can't be ported to another architecture or device because the code has been optimised and the compiler has to know the latency of the functional units.

8.2 List-based scheduling

An important part of VLIW is how the compiler schedules instructions. Scheduling is a NP problem, hence we have to use some heuristics. In particular, we are going to analyse **list-based scheduling**.

8.2.1 Dependence graph

A dependence graph is a graph that captures the dependencies between instructions. In particular,

- The nodes of the graph are the instructions of the program.
- An arc from node N to node M represents that instruction M depends on instruction N.

If we consider the following piece of code

```
i_1: add s1, s3, s3;
i_2: add s2, s1, s3;
i_3: subi s1, s4, 1;
```

we have

- A WAW dependency between instruction `i_3` and `i_1`.
- A WAR dependency between instruction `i_3` and `i_2`.
- A RAW dependency between instruction `i_2` and `i_1`.

To each arc we can associate

- A **latency**. The latency of an arc $i_a \rightarrow i_b$ represents how long it takes to execute the operation i_a.
- A **priority**. The priority of an arc $i_a \rightarrow i_b$ represents the length of the path from i_a to the end of the graph (i.e. to the node with no outgoing arcs) passing through i_b. The priority of i_a can be computed as the sum of a's latency and b's priority

$$priority_a = latency_a + priority_b$$

The dependency graph of the code above is shown in Figure 8.2.

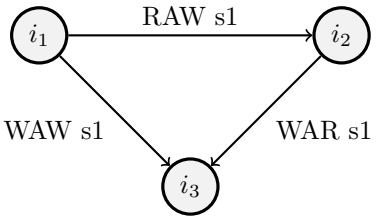


Figure 8.2: A dependency graph

8.2.2 Scheduling

A list-based scheduler uses

- A list, called **ready set**, that contains all the instructions that can be scheduled.
- A **scheduling table** with a column for each functional unit and a row for every clock cycle. The table contains what operation should occupy a functional unit in a certain clock cycle. The scheduling table can directly be mapped to the schedule.

The ready set is initialised with the input nodes of the dependency graph. The input nodes of the graph are those that don't depend on any other node (i.e., that have no incoming arc). An instruction is added to the ready set only if

- All instructions it depends on has been scheduled.
- All operands are ready.

When a compiler has to fill the scheduling table for a certain clock cycle c_i, it looks into the ready set and, for each free functional unit it checks if an instruction in the ready set can be executed on that functional unit. If more instructions can be assigned to a functional unit, the compiler chooses the one with the highest priority (i.e., the one with the longest path to the end).

8.3 Local scheduling

Local scheduling techniques allow to schedule, or more properly, reorder, the instructions of a basic block. Local scheduling techniques work together with list-based scheduling. Some important local scheduling techniques are:

- **Loop unrolling**.
- **Software pipelining**.

8.3.1 Loop unrolling

Loop unrolling allows us to replicate the body of a loop multiple times, hence doing less iteration. The basic idea (here explained in C, but that can be easily pictured in assembly) is to transform the code

```
for (int i=0; i<16; i++)
    vector[i] = vector[i] + 1;
```

in

```
for (int i=0; i<16; i+=4)
    vector[i] = vector[i] + 1;
    vector[i+1] = vector[i+1] + 1;
    vector[i+2] = vector[i+2] + 1;
    vector[i+3] = vector[i+3] + 1;
```

The number of times that the loop body is repeated is called **unrolling factor**. In our example the unrolling factor is 4.

Loop unrolling works exactly like in the dynamic case, however all optimisation are done by the compiler. Moreover, after having unrolled the loop, removed the naming dependencies and optimised it, the code can be parallelised using the list-based scheduler and issued in a bundle to obtain even higher performance.

Loop carried dependencies

To use loop unrolling, each iteration of the loop has to be independent from the others. To verify this property and correctly chose the unrolling factor, we have to consider loop-carried dependencies. Loop-carried dependencies limit the unrolling factor of loop and in some cases make it impossible. If a loop has no loop-carried dependencies, then we can choose whatever unrolling factor. Consider now the following loop

```
for(int i=5; i<20; i++) {
    vector[i] = vector[i-5] + 4;
}
```

In this case we have a loop-carried dependency because, given a value of `i`, the value written in `vector[i]` at iteration `i` depends on the value written five iterations before (because of `vector[i-5]`). This however doesn't stop us from unrolling the loop five times because with an unrolling factor of 5 we still can't experience the dependency.

```
for(int i=5; i<=16; i+=5) {
    vector[i] = vector[i-5] + 4;
    vector[i+1] = vector[i-4] + 4;
    vector[i+2] = vector[i-3] + 4;
    vector[i+3] = vector[i-2] + 4;
    vector[i+4] = vector[i-1] + 4;
}
```

Finally, if the loop-carried dependency is from one iteration to the next one, loop-unrolling can't be applied. An example is

```
for(int i=0; i<=20; i++) {
    vector[i+1] = vector[i] + 4;
}
```

because, even with an unrolling factor of 2, we would obtain

```
for(int i=0; i<=19; i+=2) {
    vector[i+1] = vector[i] + 4;
    vector[i+2] = vector[i+1] + 4;
}
```

which results in a RAW conflict.

In some cases, two loops that have a different number of iterations but a compatible body that can be merged, to reduce the loop overhead, with a technique called **peeling**. Peeling consists of merging the iterations that both cycles have to do using a loop that contains the bodies of both loops and leave out only the remaining iterations, creating a loop only for them. Consider for instance the loops

```
for(i=0;i<102;i++)
    b[i]=b[i-2]+c;

for(j=0;j<100;j++)
    a[j]=a[j]*2;
```

We can peel the two final iterations of the first loop (i.e. we create a loop ad-hoc for them) and merge the common part to obtain

```
for(i=0;i<100;i++){
    b[i]=b[i-2]+c;
    a[i]=a[i]*2;
}

for(i=100; i<102; i++) {
    b[i]=b[i-2]+c;
}
```

Advantages and disadvantages

The main advantages of loop unrolling are

- It **minimises the loop overhead**.
- **The size of the basic block increases**. This means that a scheduler can choose among more instructions.

However, this comes at a cost, in fact

- Loop unrolling **increases the register pressure** because the compiler has to use register renaming to solve naming dependencies.
- **The size of the code increases**.

8.3.2 Software pipelining

Software pipelining allows to reorganise a loop selecting and interleaving instructions from different iterations without unrolling the loop. This means that, software pipelining can be applied only if the loop has **no loop-carried dependencies** (but it can have intra-loop dependencies). In other words, we reorganize the loop in a new loop so that each new iteration, called **cycle**, executes instructions, called **stages**, chosen from different iterations of the original loop. To better explain software pipelining let us consider the following loop.

```
for(int i=0; i<N; i++){
    A[i] = B[i];
    A[i] = A[i] + 1;
    C[i] = A[i];
}
```

This loop can be rewritten, using software pipelining, as

1. A start-up code in which some instructions of the loop are executed in parallel but not all of them.
2. The for loop in which we interleave the instructions of different iterations.
3. The finish-up code in which some instructions of the loop are executed in parallel, since some iterations have already finished.

Practically we obtain

```
A[0] = B[0];

A[0] = A[0] + 1:
A[1] = B[1];

for(int i=0; i<N-2; i++){
    C[i] = A[i];
    A[i+1] = A[i+1] + 1;
    A[i+2] = B[i+2];
}

C[N-2] = A[N-2];      // C[i+0] = A[i+0]
A[N-1] = A[N-1] + 1;  // A[i+1] = A[i+1] + 1

C[N-1] = A[N-1];      // C[i+1] = A[i+1]
```

We can also try to simulate the pipelined loop execution and the result is in Table 8.1. Notice that, the x axis represents the number of iteration in the original loop while the y axis represents the execution clock cycles. All instructions on a line are called **stages** and are executed in parallel.

The big advantage in the example above is that we have an intra loop dependency (e.g., between the first and second instruction because the second uses the value of `A[i]` computed by the fisrt one), hence we can't execute in the same clock cycle all the instructions of the same loop iteration. However, since we have no loop-carried dependencies, when we execute the last instruction of the first iteration, the second of the second iteration and the first of the third iteration. Each iteration executes the instructions in order and not in the same clock-cycle, however since they are parallelised

with instructions of other iterations, we can exploit all the resources we have. This is in general the big advantage of software pipelinig.

	0	1	2	3	4
0	`A[0] = B[0]`				
1	`A[0] = A[0]+1`	`A[1] = B[1]`			
2	`C[0] = A[0]`	`A[1] = A[1]+1`	`A[2] = B[2]`		
3		`C[1] = A[1]`	`A[2] = A[2]+1`	`A[3] = B[3]`	
4			`C[2] = A[2]`	`A[3] = A[3]+1`	`A[3] = B[3]`
5				`C[3] = A[3]`	`A[3] = A[3]+1`
6					`C[3] = A[3]`

Table 8.1: The execution of a software pipelined loop with $N = 4$. The x axis represents the number of iteration in the original loop while the y axis represents the execution clock cycles.

Assembly example

To better understand how software pipelining works, let us analyse an example in assembly. Consider for instance the following loop

```
1       addi r2, r1, 400
2 LOOP: ld f1, 0(r1)
3       add f3, f1, f2
4       sd f3, 0(r1)
5       addi r1, r1, 8
6       bne r1, r2, LOOP
```

Listing 8.1: An example of non pipelined assembly code.

The code above, adds the value in register `f2` to all elements of an array of 100 elements. Leaving out the last two instructions (which are loop management instructions) and the first one (which is used to initialise the upper bound of the loop) we have to reorganise the load, the add and the store at lines 2, 3 and 4.

With software pipelining we obtain the code in Listing 8.2.

```
1       addi r2, r1, 384
2
3       ld f1, 0(r1)
4
5       add f3, f1, f2
6       ld f1, 16(r1)
7
8 LOOP: sd f3, 0(r1)
9       add f3, f1, f2
10      ld f1, 16(r1)
11      addi r1, r1, 8
12      bne r1, r2, LOOP
13
14      sd f3, 0(r1)
15      add f3, f1, f2
16
17      sd f3, 8(r1)
```

Listing 8.2: The software pipelined version of the loop in Listing 8.1

Let's analyse the software-pipelined version of the code.

Start-up The first instruction initialises `r2` with 384 bytes after the beginning of the loop, instead of 400 because parts of the last the last two iterations are done outside the loop. The second instruction executes the load for the first element of the array. The third instruction adds `f2` to the value previously loaded and the fourth loads the value in the second cell of the array in `f2`. Notice that the instructions 3 and 4 (lines 5, 6) are in reverse order with respect to the loop because otherwise the load at line 6 would overwrite the value loaded at line 3, hence we have to execute the add before loading the next value.

Loop The main loop is similar to the one in the normal version, however we notice that

- Instructions are in reverse order. This is because otherwise the load would overwrite the value that has to be used by the add, since the load refers to an iteration after the iteration of the add. The same reasoning can be applied to the store and add instructions.
- The load is done at `16(r1)` instead of `0(r1)`. Because the load refers to two iterations after the iteration of the store.

For instance, in the first iteration we want to complete the instructions done in the start up code and, not to overwrite values, we have to execute the store before. Then we execute the add for the second iteration because in the startup code we only have loaded the value of that iteration and finally we load the value for the third iteration.

Finish-up The last three instructions conclude the last two iterations. In particular, the instruction at line 14 stores the value in the last but one cell while the following instruction computes the value to put in the last cell. Note two things:

- Lines 14 and 15 can't be inverted, otherwise the `add` would overwrite the value that has to be written by instruction at line 14.
- The store is done at `0(r1)` because, we leave the loop when `r1=r2=16` and the last but one cell is 16 bytes after the first.

The last instruction is used to store the value computed at line `15`. We write the value 8 bytes after `r1` because `r1` is 384 bytes after the beginning of the array (and the last cell goes from 392 to 400 bytes after the beginning of the array).

Advantages and disadvantages

Software pipelining has many advantages with respect to loop unrolling,

- We can reduce the number of stalls.
- The instruction buffer has to store fewer instructions because the loop body is smaller.
- The pressure on the registers is reduced, because software pipelining doesn't use register renaming.

8.4 Global scheduling

Global scheduling techniques allow to schedule instruction beyond basic blocks using speculation. Some important global scheduling techniques are:

- **Trace scheduling**.
- **Super-block scheduling**.

When a speculation is wrong, the compiler can statically add some instructions, called compensation code, to rollback the wrong operations.

8.4.1 Trace scheduling

Trace scheduling allows to define the most probable path that a program will take during execution. A trace is, in fact, the most probable path followed by the program, namely, the sequence of basic blocks that it's more probable that a program will visit. A trace is build using code profiling (i.e., running the program many times) before releasing the program. Traces allow a scheduler to go beyond basic blocks to look for instructions in the trace and exploit parallelism more. In a nutshell, **trace scheduling is a compiler-generated speculation technique**.

Traces are build upon a probabilistic analysis, hence it might happen that the program doesn't execute a basic block in the trace. In such cases,

- If the processor can flush instructions, the compiler asks the processor to flush the instructions form the wrong basic block.
- If the processor can't flush instructions, the compiler has to add some code, called compensation code, to roll-back the operations of the wrong basic blocks. In this case, the size of the code increases a bit.

8.4.2 Super-block scheduling

Super-block scheduling is an extension of trace scheduling that allows the compiler to look for instructions in a super-block. A super-block is a set of basic blocks with one entrance and multiple exits. This means that, every basic block that is part of a super-block can have multiple paths: one that continues in the super-block (the most probable one) and others that go outside. An example of (already formed) super-block is shown in Figure 8.3.

Super-blocks are constructed by profiling the application and by duplicating tails (blocks after an entrance in the trace). Let us better explain this mechanism. Let's start from a program in which some blocks belonging to the trace have multiple entrances (Figure 8.4a). Since we want to obtain a trace with only one entrance point, we have to take all basic blocks after the first of the trace (in our example, from block 2 on) and, whenever we find a block B_i (e.g., block 2) with an entrance B_o (e.g., block 5) from outside the trace we

1. Duplicate block B_i (e.g., we create block $2'$) to create block B_i'.
2. We connect block B_i' to all blocks to which B_i was connected.
3. We duplicate recursively all blocks after B_i' that belonged to the trace. In our example since 2 was connected to 3 and 3 to 4, we create $3'$ and $4'$ and connect $2'$ to $3'$ and $3'$ to $4'$. Notice that, we haven't duplicated 6 nor 7 and 8 because they are not part of the trace, yet we connected $2'$ to 6, 7 and 8.

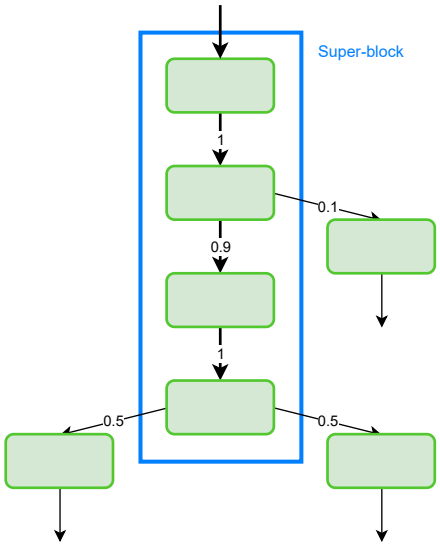


Figure 8.3: A super-block.

When all the blocks of the trace, apart from the first, have no entrance, we have obtained a super-block (Figure 8.4b). Note that, the basic blocks of the super-block can be fused together to form a single bigger block, namely, the super-block.

Advantages

The advantages of super-block scheduling are,

- Optimisation is simpler because there are no side entrances.
- We need to create compensation code only for the exits and not for the entrance.

8.4.3 Predicated instructions

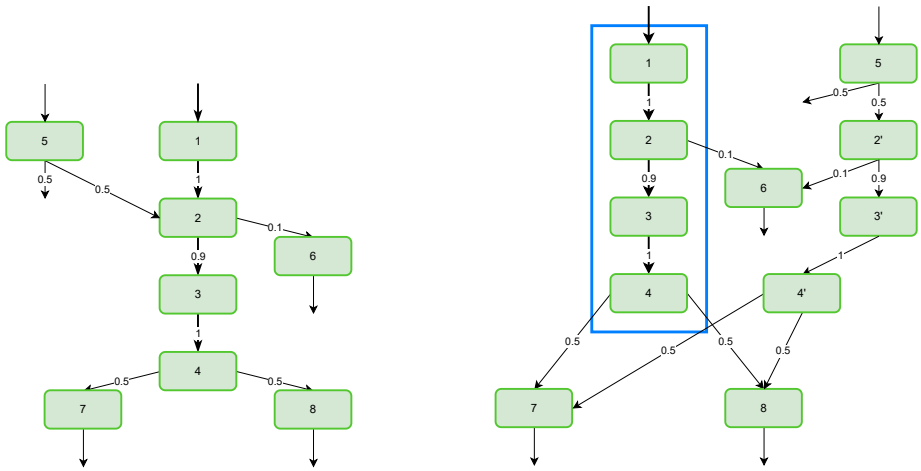
Predicated instructions allow to replace branch predictions with instructions that are committed only if a certain register, called predicate register, is true. A predicated instruction can be written as

```
(p) op r1, r2, r3
```

where

- `op` is the identifier of an operation (e.g., `add`).
- `r1`, `r2` and `r3` are normal registers.
- `p` is a Boolean register.

The operation `op` is executed only if the register `p` is true. Consider for instance the following set of instructions



(a) The basic blocks of a program before building a super-block.

(b) Duplication of basic-blocks to build a super-block.

Figure 8.4: Construction of a super-block. The path belonging to the trace is shown in bold.

```
bnez r4
add r1, r2, r3
lw r6, 0(r5)
```

which can be replaced with predicate instructions which are executed only if `r4` is equal to 0 (i.e. only if `p` is true).

```
p=(cmp r4, 0)
(p) add r1, r2, r3
(p) lw r6, 0(r5)
```

This technique is effective if,

- **Misprediction rate and penalties are considerable**. Otherwise a branch to the most likely code would result in better performance.
- **Branches are unbalanced**. The longest path is executed more frequently, otherwise we must avoid to lengthen the execution of the little and more frequent path.

Advantages

The advantages of removing branch instructions are:

- Branch misprediction is eliminated, hence the processor doesn't need to flush the pipeline.
- We enlarge the basic block size to improve scheduling.

Part III

Caches

Chapter 9

Introduction

9.1 General description

Caches are memories located close to the processor, usually implemented as Static RAM (SRAM), that are used to speed up memory access exploiting temporal and spatial locality. In particular,

- Temporal locality exploits the fact that recently accessed data might be accessed again in the near future. Data that is frequently accessed is kept in cache so that when it's accessed again, the processor doesn't have to go all the way to the memory.
- Spatial locality exploits the fact that data accessed by a program is often in contiguous pages of memory. Consider for instance a loop that iterates over an array. The cells of the array are stored in contiguous blocks of memory, hence if we load in the cache many contiguous blocks, it's probable that during a loop we don't have to access the memory but we can find all the array's cell in the cache.

9.1.1 Cache levels

Caches are divided in levels, starting from level 1 (and in most cases reaching level 3). In particular

- **L1 caches** are the closest to the CPU. L1 caches can be
  - **Unified**. Unified caches contain both data and instructions.
  - **Separated**. In this case we have two separated caches, one for data and the other for the instructions.

  If the processor is multi-core, each core has a L1 cache. L1 caches are the fastest but are also the smallest.
- **L2 caches**. L2 caches are above L1 caches and are usually unified (i.e. contain both instructions and data). L2 caches are used to reduce the miss penalty. Usually, each core has a L2 cache. L2 caches are also less fast than L1 caches but they can hold more data.
- **L3 caches** are the furthest from the CPU (still closest than main memory). Moreover, L3 caches are usually shared among multiple processors in multi-core architectures, which can create problems of cache coherency. L3 caches are the slowest caches but they can hold the largest amount of data.

In a nutshell, the more we go through high levels, the slower, larger and generic caches become. Data flows from one level of cache to the other (also considering registers and main memory as levels), this means that a L1 cache can only communicate with the registers and the L2 cache (or directly to memory if no L2 cache is present). A representation of the different levels of caches in computer is shown in Figure 9.1. L1 and L2 caches can completely share data (and instructions), or have

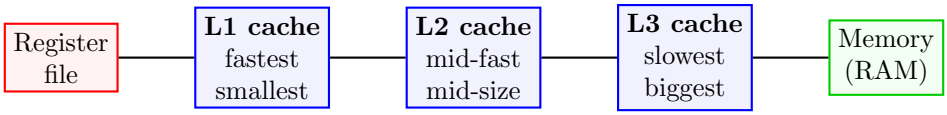
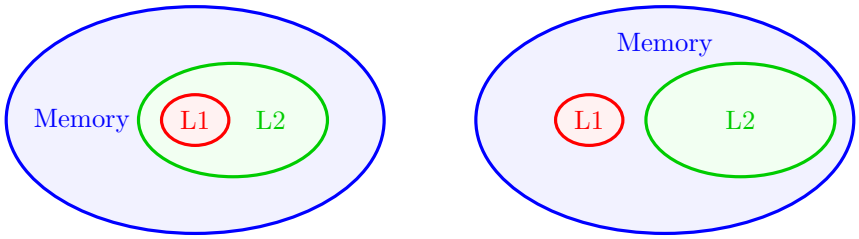


Figure 9.1: Registers, caches and memory configuration in a computer.

different values, in particular we have

- **Multi-level inclusion** if the L1 cache contains a subset of data of the L2 cache (which contains a subset of data of the memory).
- **Multi-level exclusion** if the L1 and L2 caches don't share data.



(a) Multi-level inclusion. (b) Multi-level exclusion.

Figure 9.2: Ways of sharing data between L1 and L2 caches.

9.1.2 Memory interface

All cache levels are on the same chip and are implemented using SRAM, while the main memory is a separated component and it's implemented using DRAM. This means that we need an interface to allow cache and main memory to communicate. Also remember that accesses to SRAMs cost much less than accesses to DRAMs.

9.1.3 Data

Data in a cache is organised in blocks, which are multiple of words. This means that the minimum chunk of data that can be moved is a block. The reason of this choice is to exploit data locality, in fact if the processor is accessing a word, it might probably need the next one (e.g. after accessing the first 4 chars of a string we need to access the next 4). Given the size S_{cache} of the cache and the size S_{block} of a block we can compute the number N_{blocks} of cache blocks in a cache as

$$N_{blocks} = \frac{S_{cache}}{S_{block}}$$

9.1.4 Cache access

When the processor needs to access a datum in memory, it access the cache first because it only costs 1 clock cycle. The cache access can generate two events

- A **cache hit**. A cache it verifies when the block needed by the processor is in the cache (even in the upper levels). In this case the block is returned and the operation completes in 1 clock cycle.
- A **cache miss**. A cache miss verifies when the block needed by the processor is not in the cache (considering all levels). In this case the processor has to stall because it needs to retrieve the block from main memory, which costs multiple clock cycles. Schematically, the processor has to
  1. Stall the CPU.
  2. Get the block from main memory.
  3. Write the block in the cache.
  4. Repeat the cache access.

Notice that, a cache miss happens both when the processor wants to read or write something. In particular,

- When the processor wants to **read**, we have a read from cache (by the processor) and a write to cache (by memory that writes the block).
- When the processor wants to **write**, two write operations are executed, one by the processor and one by the memory.

9.1.5 Performance

To evaluate the performance of a cache we need some indicators.

Hit rate

The hit rate f_{hit} is frequency at which the processor finds the block it's looking for in the cache and it's computed as the ratio between the number of cache hits $N_{cache\ hits}$ and the number of memory accesses $N_{memory\ accesses}$.

$$f_{hit} = \frac{N_{cache\ hits}}{N_{memory\ accesses}}$$

Hit time

The hit time t_{hit} is the time required to access a block of data in the upper level cache following a cache hit. t_{hit} can be expressed in number of clock cycles or in seconds (a common value is 1 ns).

Miss rate

The miss rate f_{miss} is the frequency at which the processor doesn't find a block in the cache and it's computed as the ratio between the number of cache misses $N_{cache\ misses}$ and the number of memory accesses $N_{memory\ accesses}$.

$$f_{miss} = \frac{N_{cache\ misses}}{N_{memory\ accesses}}$$

Miss penalty

The miss penalty $t_{penalty}$ is the time needed to access the lower level of cache and replace the block in the upper level of cache. Usually the miss penalty is much larger than the hit time (i.e., $t_{hit} \ll t_{penalty}$).

Miss time

The miss time t_{miss} is the sum of the hit time (when the cache is accessed again after the miss) and the miss penalty (that accounts for the processor not finding the data and retrieving it from memory).

$$t_{miss} = t_{hit} + t_{penalty}$$

Average Memory Access Time

The Average Memory Access Time $AMAT$ is the weighted average between the hit time and the miss time.

$$AMAT = f_{hit} \cdot t_{hit} + f_{miss} \cdot t_{miss}$$

The miss time can be replaced using $t_{miss} = t_{hit} + t_{penalty}$ to obtain

$$\begin{aligned} AMAT &= f_{hit} \cdot t_{hit} + f_{miss} \cdot t_{miss} \\ &= f_{hit} \cdot t_{hit} + f_{miss} \cdot (t_{hit} + t_{penalty}) \\ &= f_{hit} \cdot t_{hit} + f_{miss} \cdot t_{hit} + f_{miss} \cdot t_{penalty} \\ &= (f_{hit} + f_{miss}) \cdot t_{hit} + f_{miss} \cdot t_{penalty} \end{aligned}$$

By definition, the sum of miss and hit rate is 1, hence we can rewrite the average access time as

$$AMAT = t_{hit} + f_{miss} \cdot t_{penalty}$$

This formula highlights the parameters we can try to optimise to reduce the average memory access time, i.e. hit time, miss penalty and miss rate.

Unified caches If we consider unified caches, the formula to compute the $AMAT$ has to be modified a bit to consider the fact that the cache is accessed for data and instruction, which have different hit rates. In particular we obtain

$$AMAT = \frac{N_{data}}{N_{total}} \cdot (t_{hit} + f_{miss,data} \cdot t_{penalty}) + \frac{N_{instruction}}{N_{total}} \cdot (t_{hit} + f_{miss,instruction} \cdot t_{penalty})$$

where

- N_{data} is the number of accesses to get data.
- $N_{instruction}$ is the number of accesses to get instructions.
- N_{total} is the total number of accesses to the cache.

Also notice that usually, the miss rate of data accesses is much higher than the one of instruction accesses because instructions are usually stored in contiguous areas of memories, hence the cache can better exploit spatial locality.

L2 and L3 caches If we consider L2 and L3 caches, the $AMAT$ has to be refined replacing the miss time with the access time to the upper level cache. In particular the miss penalty $t_{miss,L1}$ at level $L1$ in

$$AMAT = t_{hit,L1} + f_{miss,L1} \cdot t_{penalty,L1}$$

can be replaced with the $AMAT$ of the L2 cache to obtain

$$\begin{aligned} AMAT &= t_{hit,L1} + f_{miss,L1} \cdot t_{penalty,L1} \\ &= t_{hit,L1} + f_{miss,L1} \cdot (t_{hit,L2} + f_{miss,L2} \cdot t_{penalty,L2}) \\ &= t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1} \cdot f_{miss,L2} \cdot t_{penalty,L2} \end{aligned}$$

The product between the miss rate of the L1 and L2 caches is called global miss rate and represents the frequency at which a data isn't found neither in the L1 cache nor in the L2 cache. If we call $f_{miss,L1L2}$ the global miss rate we obtain the following formula to compute the average memory access time

$$AMAT = t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1L2} \cdot t_{penalty,L2}$$

The same procedure can be repeated with the L3 cache, hence we can replace $t_{penalty,L2}$ with the $AMAT$ of L3 and obtain

$$\begin{aligned} AMAT &= t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1L2} \cdot t_{penalty,L2} \\ &= t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1L2} \cdot (t_{hit,L3} + f_{miss,L3} \cdot t_{penalty,L3}) \\ &= t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1L2} \cdot t_{hit,L3} + f_{miss,L1L2} \cdot f_{miss,L3} \cdot t_{penalty,L3} \\ &= t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1L2} \cdot t_{hit,L3} + f_{miss,L1L2L3} \cdot t_{penalty,L3} \end{aligned}$$

where, as before, $f_{miss,L1L2L3} = f_{miss,L1} \cdot f_{miss,L2} \cdot f_{miss,L3}$.

9.2 Cache structure

Every entry in a cache memory is made of

- A **valid** bit. The valid tells if the entry is valid.
- A **dirty** bit. The dirty tells if the entry has been written.
- A **tag**. The tag uniquely identifies the memory address of the data written in the entry.
- The **data**. The data contains a block (i.e. multiple words) of memory.

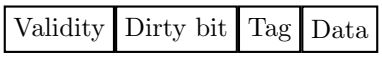


Figure 9.3: An entry in a cache memory.

Furthermore, there exists three main architecture to build caches,

- **Direct-mapped** caches.
- **N-ways set-associative** caches.
- **Fully associative** caches.

9.2.1 Direct mapped caches

In direct mapped caches, a block of memory at address A can be stored in one and one only cache entry. To decide in which cache address the block should be putted we have to compute the modulo between the memory address and the number of blocks in the cache.

$$addr_{cache} = addr_{mem} \mod N_{blocks\ in\ cache}$$

Before understanding how the processor obtains the correct value from the cache we have to divide the memory address in

- T most significant bits, called **tag**.
- I middle bits, called **index**.
- $O = W + B$ least significant bits called **offset**.

Where the sum of T, I and O is the total length of the address. The different components of the

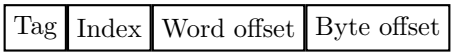


Figure 9.4: A memory address in direct mapped caches.

memory address are used as follows

- The **tag**, which is present in the cache entry, is used to check if the block in cache is the same of the block in memory. Remember that the cache is much smaller than the memory, hence multiple memory addresses are mapped to the same cache entry. Notice that the tag doesn't indicate what block should we look into, it's only used to check that the block is actually the one we are looking for.
- The **index** indicates the block of cache (i.e. the cache address, the cache entry) where the data is. Notice that the length of the index has to be equal to n if the cache has 2^n entries.
- The **offset** indicates the word or the byte that have to be taken from the block of data. We can have both the word and the byte address.

Algorithmically (even if the operations are executed in parallel), the processor

1. Takes the cache entry at the address specified by the index (i.e. computes the modulo between memory address and the number of blocks).
2. Checks if the validity bit is set to 1.
3. Checks if the tag of the entry is equal to the tag of the memory address.
4. Takes the byte of the data block specified by the offset.

Notice that in this case we need only one comparator, in the sense that we need only to check one entry of the cache. A representation of a direct mapped cache is shown in Figure 9.5.

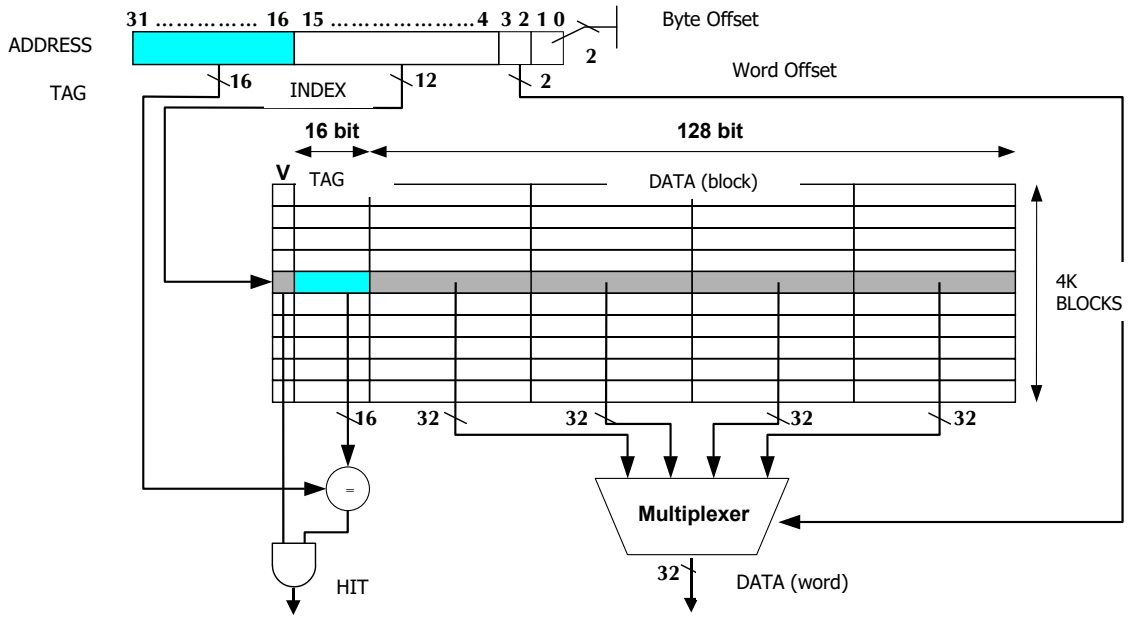


Figure 9.5: A direct mapped cache.

9.2.2 Fully associative cache

In fully associative caches, a block can be putted whenever the processor wants. In practice, the processor uses a policy to decide where to put a data block to maximise locality, but in general it's free to use whatever block it wants. This means that when we want to obtain a block from the cache we have to check all the entries because, in principle, a block can be wherever it wants. Comparing this architecture with direct-mapped caches we notice immediately that this one uses much more comparators than the former one. Since in this architecture we have no mechanism to decide where to map a memory address, the address should be interpreted differently with respect to direct mapping. In particular, an address is divided in

- T most significant bits, i.e. the **tag**.
- $O = W + B$ least significant bits, i.e. the word and byte **offset**.

Figure 9.6: A memory address in fully associative caches.

In this case, when the processor want to retrieve some data from cache, it has to

1. Check in parallel all the cache entries and verify if one entry has the same tag of the memory address of the data.
2. Take the byte of word specified by the respective offsets.

An example of fully associative cache is shown in Figure 9.7.

Block allocation

In fully associative caches, the processor can choose to put a block wherever it wants in the cache. In particular, the processor can implement three different policies to choose where to put a memory block,

- **Random**. The random policy says that the processor should put the block in a random position.
- **Last Recently Used** (LRU). The LRU policy says that the processor should replace the block that hasn't been used for the longest time (i.e. the last recently used).
- **FIFO**. The FIFO policy says that the processor should replace the oldest block (i.e. the one inserted first in the cache). Notice that, this policy is different from LRU, in fact, the oldest block might be frequently used, hence some other block might be the last recently used.

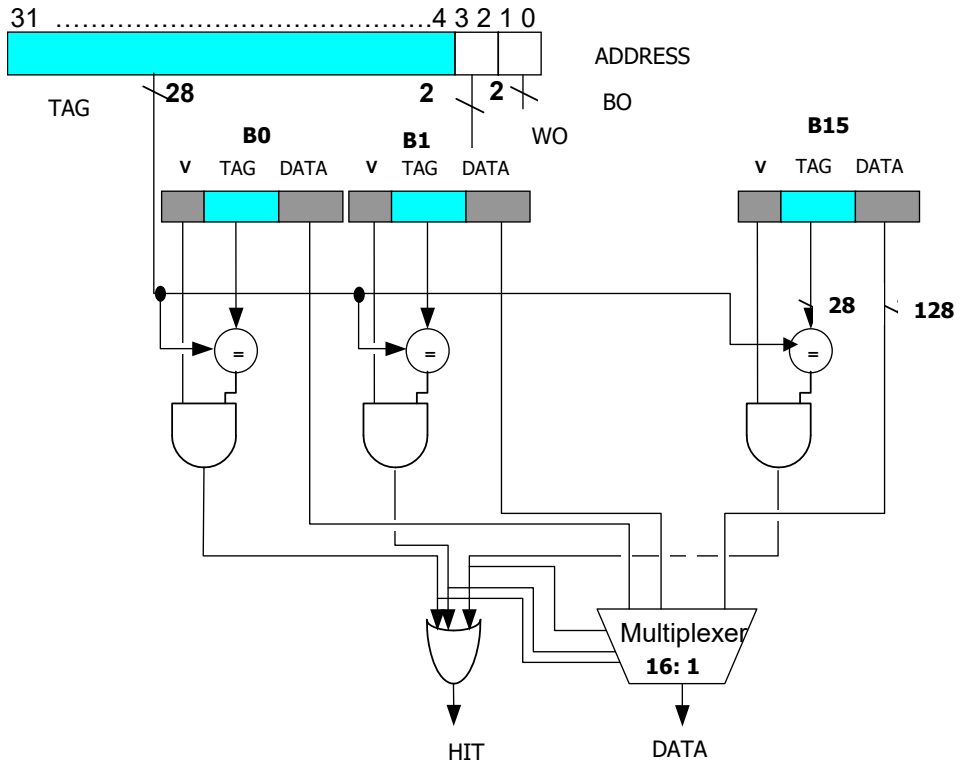


Figure 9.7: A fully associative cache.

9.2.3 N-ways set associative caches

N-ways set associative caches are an hybrid between direct mapped caches and fully associative caches. In particular they divide the cache in sets of blocks, mapped using direct mapping. Each set contains N blocks and works like a fully associative cache, namely, the cache is fully associative

only inside each set. Notice that the number N represents the number of ways (i.e. the number of blocks in each set) and not the number of sets in which the cache is divided. Before diving into the workings of this architecture, let us divide the memory address in

- T most significant bytes, i.e. the **tag**.
- I bits, i.e. the **index**
- $O = W + B$ least significant bits, i.e. the word and byte **offsets**.

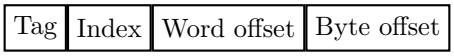


Figure 9.8: A memory address in N-ways set associative caches.

The processor

1. Uses the index (i.e. computes the modulo operation like in direct mapped caches) to check in which set to look.
2. Uses the tag to check, in parallel, if one of the entries of the set has the same tag of the memory address of the data.
3. Checks if the validity bit is set to 1.
4. Uses the word and byte offsets to retrieve the correct word or byte.

An example of N-way set associative cache is shown in Figure 9.9.

9.3 Handling write operations

9.3.1 Write misses

On a write miss (i.e. when the processor wants to write a block which is not in the cache), the processor can actuate two different policies to handle the write operation,

- **Write through**. The write through policy says that the processor should write the data in the cache and then immediately in main memory.
- **Write back**. The write back policy says that the processor should write the data only in the cache.

Write through

The write through policy says that the processor should write the data in the cache and then immediately in main memory. When the processor has to write a block to memory it can

- Write the block **directly to memory** and pay the miss penalty.
- Write the block **to a write buffer**. The write buffer is implemented on the chip, hence the processor can access it swiftly and it doesn't has to pay the miss penalty. The buffer has an interface with the main memory and will asynchronously handle the actual write in memory. Notice that the buffer should be long enough not to saturate it, however this isn't a problem in modern processors.

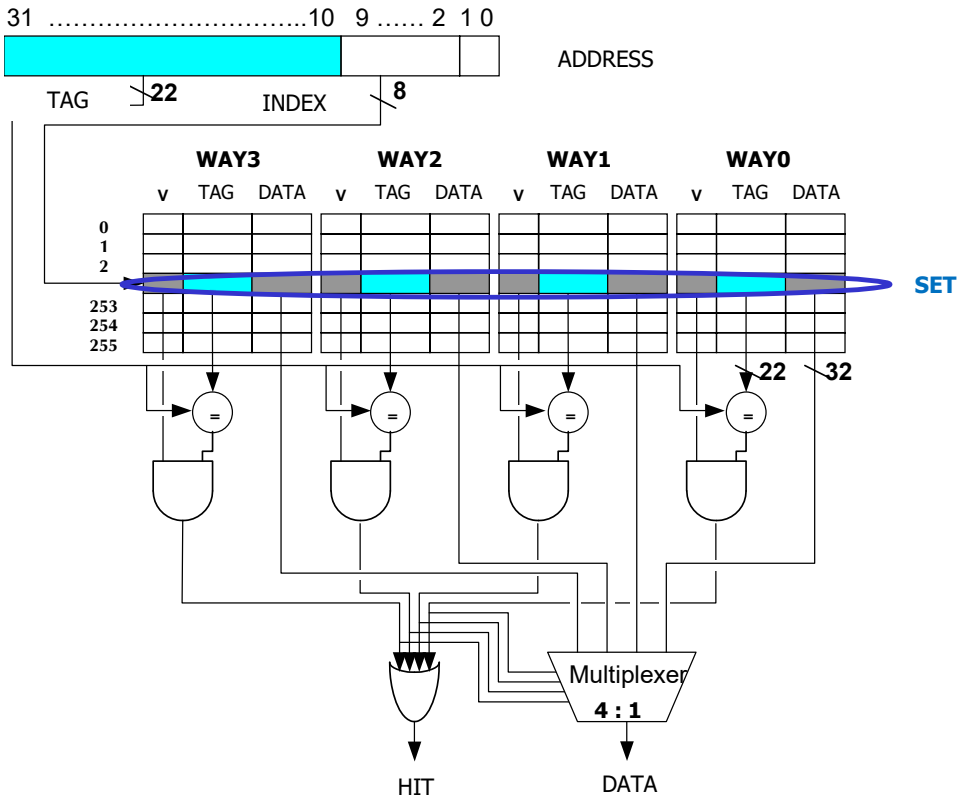


Figure 9.9: An 4-ways set associative cache.

The main advantages of this policy are

- It's simpler to implement, but to be effective it requires a write buffer to do not wait for the lower level of the memory hierarchy (to avoid write stalls).
- The read misses are cheaper because they do not require any write to the lower level of the memory hierarchy.
- Memory is always up to date.

Write back

The write back policy says that the processor should write the data only in the cache. The block is written in memory only when it's replaced by another block. The dirty bit is used exactly to implement this policy, in fact when a block is replaced, the processor checks if the dirty bit is set to 1 and if so, the block is written to main memory. The main advantages of this policy are

- The block can be written by the processor at the frequency at which the cache, and not the main memory, can accept it.
- Multiple writes to the same block require only a single write to the main memory.

9.3.2 Option write allocate

After a write miss, the processor also has to decide whether to allocate a new block in cache or not. This behaviour is enforced by two policies

- **Write allocate**. The write allocate policy tells the processor to allocate a new block in cache, write the value in memory and write the new value.
- **No write all**. The no write all policy tells the processor to send the data directly to memory (either directly or through the write buffer) without reserving a buffer entry.

The write allocate option can be combined with the write through/back policy to obtain different behaviour on write misses.

Chapter 10

Performance improvements

Before analysing the ways we can improve cache access performance, we should characterise the types of cache misses that can happen. In particular, cache misses are divided in

- **Compulsory misses**. Compulsory misses are inevitable misses and are caused by the fact that, at startup, the memory is randomly initialised, hence we will surely have cache misses the first time an entry is accessed. More precisely, when the processor boots up, the validity bit of the cache is set to 0 for each entry so that the first access to each entry surely results in a cache miss. This types of misses are independent from the cache size, however, the bigger the cache, the more misses we have because we have to replace more blocks. In other words, having a bigger cache, doesn't impact on the number of misses for each entry.
- **Capacity misses**. Capacity misses are generated by the fact that the cache is smaller than the memory, hence some blocks in cache have to be replaced by others when needed. This type of misses is more rare as the memory size increases because we have more space to accommodate the blocks.
- **Conflict misses**. Capacity misses are generated by the fact that the cache is smaller than the memory, hence more memory blocks are mapped to a cache entry. This type of misses only verifies in direct mapped and set-associative caches because fully-associative caches don't use mapping. Conflict messes become more rare as associativity increases and are avoided in fully-associative caches.

Now that we know how cache misses are classified, we can focus on how to reduce the cache access time. If we consider the formula to compute the $AMAT$

$$AMAT = t_{hit} + f_{miss} \cdot t_{penalty}$$

we notice that we can reduce three different parameters,

- The hit time t_{hit}.
- The miss rate f_{miss}.
- The miss penalty $t_{penalty}$.

10.1 Reducing the miss rate

The miss rate can be reduced using

- **Dynamic and hardware-based techniques**.
- **Static techniques** applied by the compiler.

10.1.1 Dynamic and hardware-based techniques

Area increase

The first method to reduce the miss rate is to increase the capacity (i.e. the size) of the cache. This technique definitely decreases the miss rate, however it increases the area of the CPU, hence its power consumption and its cost.

Changing the block size

Another method to reduce the miss rate is to increase the size of block (without changing the size of the memory). In other words, we have less entries that can store more data. This technique reduces the compulsory misses because we have less entries (still the same cache size), however it increases the miss penalty and the conflict misses because we have less entries, hence more memory addresses are mapped to the same cache entry. This means that we always have to consider a trade-off between the number of entries and the block size to get the best performance possible.

Increase associativity

Increasing the cache associativity (i.e. moving from direct mapping to fully-associative caches) can decrease the conflict misses, hence reducing the miss rate. However, associative caches require more hardware because every entry has to be confronted with the memory address of the block to retrieve. Having more hardware means having a bigger chip area, bigger power consumption and higher costs. To find a good trade-off between associativity and hardware complexity we can consider that (empirically) a cache of size N has almost the same miss rate of a 2-way set-associative cache of size $\frac{N}{2}$.

Simulating associativity Associativity can also be simulated using interleaved memories. More precisely, the cache is divided in independent banks that support simultaneous access to increase the bandwidth (because we can access all the banks in parallel).

Victim cache

The miss rate can also be reduced introducing a small cache, called victim cache, between the L1 and L2 caches. The victim cache collects the replacements of the L1 cache and on a miss, the processor can look into the victim cache to check if the searched block is there. Basically, the victim cache tries to exploit temporal locality. The main characteristics of the victim cache are

- It's **very small**.
- It's **fully associative**.
- It **stores recently discarded data**.

Way prediction and pseudo-associativity

In N-way set-associative caches, the processor can try to predict which way a request will go to to speed up memory access. In particular,

- If the **prediction is correct** (and the block is in the cache), the processor will pay the **hit time**.
- If the **prediction is wrong** (and the block is in the cache), the processor will pay a **pseudo-hit time** which is bigger than the hit time because considers the fact that the processor has to access a way different from the predicted one.
- If the block is **not in memory**, the processor will pay the **miss penalty**.

The same predicting behaviour can be applied to pseudo-associative direct-mapped caches (i.e. direct-mapped cache that simulate associativity with interleaved banks) but in this case the processor has to predict the correct bank. As before the processor pays either the hit time, the pseudo-hit time or the miss penalty.

Notice that, in both cases, the prediction can be computed either dynamically by the processor or statically by the compiler.

Hardware pre-fetching

The last technique used to decrease the miss rate is pre-fetching. This optimisation consists in putting a stream buffer close to the cache (i.e. on the CPU) and when an instruction or data block is copied from memory to the cache (due to a miss), the successive block is copied to the buffer. Basically, the processor is pre-fetching the next set of instructions or the next data block so that, if needed, it can be accessed and copied to cache much faster because the buffer is close to the cache and on the CPU. This technique is very effective at exploiting spatial locality, however it requires extra memory bandwidth because we need to move more data from the memory to the processor. Notice that, pre-fetching can also be applied statically at the software level.

10.1.2 Compiler optimisations

Compilers can recognise some situations in which some cache optimisations can be applied. As always, these techniques are very effective but they rely on the compiler and on the fact that it's smart and that it knows the structure of the cache. All the techniques applied by compilers come from software profiling.

Array merging

Merging arrays improves spatial locality. Say we have two arrays of the same length, we can create a structure that stores one element of the first array and one of the second. Thanks to this structure we can create a new (merged) array of this structure. Having merged the two arrays, if we have to access cells at the same position in the two arrays, we don't have to load data from two different arrays (risking to generate misses and continuous replacements) because both cells are in the same structure, hence in the same block. An example of a merged array obtained from two source arrays is shown below.

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of structures */
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];
```

Loops interchange

Loops interchange swaps inner and outer loops to exploit spatial locality and reduce the continuous replacement of blocks (called striving). In this case we can see how important is that the compiler knows the cache structure, in fact it has to know how a data structure (e.g. a matrix) is stored in memory. An example of loops interchange that exploits spatial locality in a matrix is shown below.

```
/* Before */
for (j = 0; j < 100; j = j+1) {
    for (i = 0; i < 5000; i = i+1)
        x[i][j] = 2 * x[i][j];
}

/* After */
for (i = 0; i < 5000; i = i+1) {
    for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];
}
```

Loop fusion

Loop fusion improves spatial locality by combining 2 independent loops (i.e. their body) that have same looping and some variables overlap. An example of loop fusion is shown below.

```
/* Before */
for (i = 0; i < N; i = i+1) {
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
}
for (i = 0; i < N; i = i+1) {
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
}

/* After */
for (i = 0; i < N; i = i+1) {
    for (j = 0; j < N; j = j+1) {
        a[i][j] = 1/b[i][j] * c[i][j];
```

```
        d[i][j] = a[i][j] + c[i][j];
    }
}
```

Loop blocking

Loop blocking improves temporal locality by accessing sub-blocks of data repeatedly instead of accessing by entire columns or rows. Say we have a matrix, we can divide it in $B \cdot B$ (B is called blocking factor) sub-matrices that fit in a block and so that we can load one block in cache and do all the accesses in one time.

Part IV

Beyond instruction level parallelism

Chapter 11

Introduction

Instruction level parallelism allow to speed up a processor's throughput, however, due to cache misses and branch mispredictions, the real CPI is far from ideal. To further improve the CPI we can apply

- **Multi-threading**. Multi-threading allows to execute, on a single processor, also called core, multiple threads in parallel.
- **Multi-core chips**. Multi-core chips are chips made of multiple processors, or cores, connected together.

Note that, the concepts of multi-threading and multi-coring are orthogonal. This means that, one can have

- A chip with one core and multiple threads.
- A chip with multiple cores that execute one thread only.
- A chip with multiple cores, each executing multiple threads.

An exhaustive depiction of the combinations of multi-coring and multi-threading is shown in Table 11.1.

	Single-core	Multi-core
Single-thread	Single-core Single-thread	Multi-core Single-thread
Multi-thread	Single-core Multi-thread	Multi-core Multi-thread

Table 11.1: All the possible combinations of cores and threads on a chip.

Chapter 12

Multi-threading

12.1 Threads

Threads are lightweight processes that share the memory with the parent process (i.e., the process that created them). Threads are more lightweight than processes because, since they share the memory of their parent, the processor doesn't have to switch context every time a thread replaces a running thread.

Threads can be explicitly handled by the programmer or implicitly used by the processor to improve performance. For instance, in the latter case, a processor might assign every loop iteration (of a loop without carried-dependencies) to a different thread to maximise parallelism.

12.1.1 Hardware modifications

Multi-threading can be applied to processors with single and multiple functional units. In both cases, we need to add some hardware to handle multiple threads of execution, in particular if we want to support N threads, we need

- **One program counter for each thread**, hence N program counters.
- **One register file for each thread**, hence N register files. A multiplexer is used to take a value from the correct register file. In general, the processor can also have a single RF, however it should be partitioned so that every thread has its own registers.
- **Control hardware** to handle thread changes (which are much less expensive than context switches).

In other words, the functional units can be shared among threads but register files and program counters have to be independent (one per thread).

12.2 Multi-thread processors

Superscalar processors (i.e., multi-issue processors with multiple FUs) can support different types of multi-threading:

- **Coarse-grained multi-threading**.

- **Fine-grained multi-threading**.
- **Simultaneous multi-threading** (SMT).

Despite their internal differences, every multi-threading model is based on the fact that, when a superscalar processor tries to parallelise some instructions, it has to search in small basic blocks and it has to solve many data dependencies. This means that the code is full of `nops` and some functional units aren't used. The idea is to fill the empty slots of the functional units with instructions of other threads and improve the IPC. Each type of multi-threading has different policies to fill the empty spaces.

In all cases, multi-thread processors pick N threads that can be executed in parallel and schedule them switching from one to the other. When a thread finishes its execution, a new thread is picked.

12.2.1 Coarse-grained multi-threading

Coarse-grained multi-threading processors execute the code of another thread when the thread in execution is blocked due to stalls. In this case, all resources are used by the second thread, hence all functional units are monopolised by a thread. An example of coarse-grained multi-threading is shown in Figure 12.1.

	FU1	FU2	FU3	FU4
Cycle 1	Thread 1	Thread 1	Thread 1	Thread 1
Cycle 2	Thread 1	Thread 1		
Cycle 3	Thread 2	Thread 2		
Cycle 4	Thread 2	Thread 2		
Cycle 5	Thread 3	Thread 3	Thread 3	Thread 3
Cycle 6	Thread 3	Thread 3	Thread 3	Thread 3
Cycle 7	Thread 1	Thread 1		

Table 12.1: Execution of three threads on a coarse-grained multi-threaded processor with four functional units.

12.2.2 Fine-grained multi-threading

Fine-grained multi-threading processors (can) switch threads at each clock cycle. More precisely, at each clock cycle the processor checks if it can continue the execution for the current thread or it should execute another thread. As for coarse-grained multi-threading, the functional units are used only by a thread in each clock cycle. An example of fine-grained multi-threaded is shown in Figure 12.2.

Advantages and disadvantages

Fine-grained multi-threading processors need more hardware because, at each clock cycle, the processor can switch thread, however, we can achieve **more parallelism**.

An important thing to notice it that, fine-grained multi-threading brings an advantage to the program as a whole and not to a single thread. Ideally, a program with N functional units has a CPI of

$$CPI_{ideal,single} = \frac{1}{N}$$

	FU1	FU2	FU3	FU4
Cycle 1	Thread 1	Thread 1	Thread 1	Thread 1
Cycle 2	Thread 2	Thread 2		
Cycle 3	Thread 1	Thread 1		
Cycle 4	Thread 3	Thread 3		
Cycle 5	Thread 3	Thread 3	Thread 3	Thread 3
Cycle 6	Thread 2	Thread 2	Thread 2	Thread 2
Cycle 7	Thread 1	Thread 1		

Table 12.2: Execution of three threads on a fine-grained multi-threaded processor with four functional units.

However, since we have M threads, a thread has to share the resources (i.e., the FUs) with other $M-1$ threads, hence the CPI is reduced to

$$CPI = \frac{M}{N}$$

Moreover, the real CPI is greater than the ideal $CPI_{id} = \frac{1}{N}$ because of branch mispredictions and cache misses, hence there isn't that much of a difference between the real CPI and the multi-threaded CPI. The real CPI can also be reduced because, while other threads are in execution, some conflicts might solve automatically.

12.2.3 Simultaneous multi-threading

Simultaneous multi-threading processors can execute different threads in the same clock cycle. This means that the functional units are shared, in the same clock cycle, among multiple threads. More precisely, at each clock cycle, the processor checks which threads can execute some instructions, pooling them in a round-robin-fashion. As for the other types of multi-threading techniques, the processor still has to handle cache misses and branch mispredicitons.

	FU1	FU2	FU3	FU4
Cycle 1	Thread 1	Thread 1	Thread 2	Thread 2
Cycle 2	Thread 1	Thread 3		
Cycle 3	Thread 2	Thread 2		
Cycle 4	Thread 1	Thread 2		
Cycle 5	Thread 3	Thread 3	Thread 1	Thread 2
Cycle 6	Thread 1	Thread 1	Thread 3	Thread 2
Cycle 7	Thread 1	Thread 2		

Table 12.3: Execution of three threads on a simultaneous multi-threaded processor with four functional units.

Performance

Simultaneous multi-threading doesn't only reduce the number of empty slots, hence maximising throughput, but it also reduces power consumption.

Multi-core and multi-core chips

If we consider a simple processor, say a dual issue one, we can apply simultaneous multi-threading and at the same time put more processors (also called cores) in the same chip, since a simple processor is quite small. This allows to build a processor that

- Is **multi-core**, each core having a shared cache.
- Uses **multi-threading**.
- Has **multi-issue**.

The number of cores, threads and issues can change, depending on the application for which the chip is thought.

Chapter 13

Multi-core processors

Multi-core chips are chips on which multiple cores are connected one to the other to increase the processing power. To connecting more cores together we have to define,

- How many cores should be putted on the same chip.
- How powerful the cores should be.
- How to connect the cores.
- How to store data and where to place the physical memory.
- How cores should be synchronised.
- How to program cores.
- How to maintain cache coherence.
- How to evaluate performance.

13.1 Core connection

In a multi-core processor, the cores should be somehow connected so that they can share some information. Core connections can be divided in

- **Single bus**.
- **Interconnected networks**.

In both cases, we will refer to nodes as the units that should be connected one to the other. A node can be a core (with its low level cache), a memory device or an I/O device. When evaluating which of the two solutions is better, in relationship with the number of cores, we should consider performance, cost and the ratio between the two.

13.1.1 Single bus connection

In a single bus topology, each node is connected to the same busses (one bus for data bus and one for instructions). This means that, only one node at a time can use a bus and if a node wants to use the bus which is occupied, it should wait until it's free.

13.1.2 Interconnected networks

In interconnected networks, a node is made of a core, its low level caches and a piece of main memory. This means that, the main memory is partitioned between the different nodes. In interconnected networks, each node is connected to one or more nodes via a direct and exclusive link. This means that each link can be used independently and each node can communicate simultaneously with multiple other nodes.

Nodes can be connected in different topologies. To analyse the performance of a topology, we should consider its bandwidth. In particular, we should consider

- The **maximum bandwidth** B_{max}, which is the best case bandwidth.
- The **bisection bandwidth** B_{bis}, which is the bandwidth of the set of links that divides the network in two identical parts. The bisection bandwidth is a standard approximation of the worst case bandwidth.

When computing these values for each topology we will call

- N the number of nodes of the network.
- b the bandwidth of a single link.

Ring topology

In the ring topology, nodes are placed in a loop where each node is connected to two other nodes. A graphical example of a ring topology is shown in Figure 13.1. In the best case scenario (i.e., when every node sends a packet to its successor on the left), the maximum bandwidth is N times the bandwidth of a link.

$$B_{max,ring} = N \cdot b$$

The bisection bandwidth is, instead, 2 times the bandwidth of a link because the bisection crosses two links (as we can see in Figure 13.1).

$$B_{bis,ring} = 2 \cdot b$$

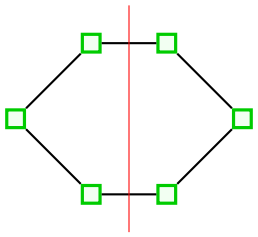


Figure 13.1: A ring topology.

Crossbar topology

In the crossbar topology, each node is connected to every other node. A graphical example of a crossbar topology is shown in Figure 13.2. In the best case scenario, i.e., when all links are used at

the same time, the total bandwidth is $\frac{(N-1)^2+N-1}{2}$ times the bandwidth of a link, because we have $\frac{(N-1)^2+N-1}{2}$ links.

$$B_{max,crossbar} = \frac{(N-1)^2 + N - 1}{2} \cdot b = \frac{N \cdot (N-1)}{2} \cdot b$$

The bisection bandwidth is, instead

$$B_{max,crossbar} = \frac{N^2}{4} \cdot b$$

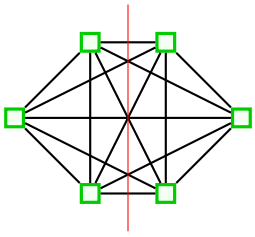


Figure 13.2: A crossbar topology.

Mesh topology

In the mesh topology, nodes are placed in a grid and each node, apart from those on the borders, is connected to 4 nodes: one north, one east, one south and one west. A graphical example of the mesh topology is shown in Figure 13.3. The maximum bandwidth for the mesh topology is obtained summing the number of vertical and horizontal links (both $\sqrt{N}(\sqrt{N}-1)$).

$$B_{max,mesh} = \sqrt{N}\big(\sqrt{N}-1\big) + \sqrt{N}\big(\sqrt{N}-1\big) \cdot b = 2\sqrt{N}\big(\sqrt{N}-1\big) \cdot b$$

From Figure 13.3 we understand that, the bisection bandwidth is

$$B_{bis,mesh} = \sqrt{N} \cdot b$$

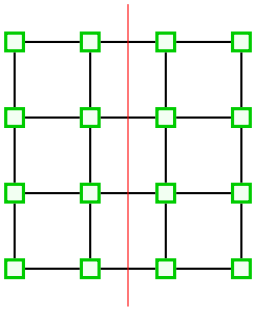


Figure 13.3: A 2D-mesh topology.

The mesh topology can also be extended to multiple dimensions to obtain an **hypercube topology**.

Torus

The torus topology is an evolution of the mesh topology. In particular, each node on the border is connected to the respective node on the other border. A graphical example of the torus topology is shown in Figure 13.4. The bisection bandwidth of the torus topology is

$$B_{bis,torus} = 2\sqrt{N} \cdot b$$

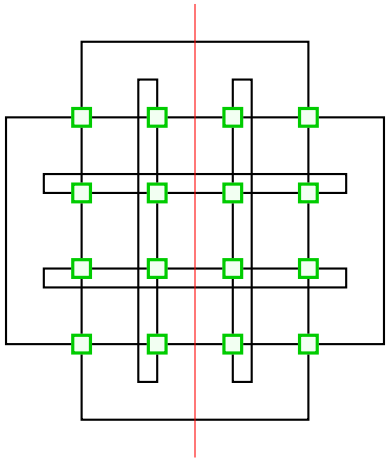


Figure 13.4: A torus topology.

13.2 Data sharing and physical memory location

13.2.1 Memory Address Space models

The main memory can be divided into chunks called modules. Each module can belong to one or more cores and, depending on which processes each module belongs to, we have different Memory Address Space models. In particular, we recognise

- The **Single Logically Shared** (SLS) address space. In the SLS address space model, all modules belong to all cores. This means that every core can load and store data to and from each memory module. This implies that cores can exchange data writing and reading on the same module. In SLS architectures, the same physical address on 2 processors refers to the same location in memory. The shared memory model generates cache coherence problems because the processors access to the same memory modules.
- The **Multiple and Private** (MP) address space. In the MP address space model, each core is assigned a unique set of modules, hence a core can't load and store data from and to a module which belongs to another core. Data is exchanged using messages. More precisely, a core $C1$ can send a request message to a core $C2$ asking for a data that is on $C2$'s memory and $C2$ replies with the requested data. For this reason, this model is also called **Message Passing** model. In MP architectures the same physical address on 2 processors refers to 2 different locations in 2 different memory modules. In this case, since every process accesses a different set of modules, we have no cache coherency problems.

Memory Address Space models are independent from the network topology and from the location of the physical memory. This means that the models are only logical models that describe who can access some modules of memory, independently from where they are physically.

13.2.2 Physical memory location

Independently from the Memory Address Space model, the physical memory can be

- **Centralised** if it's **placed on a node of its own** like in bus topologies. This memory location policy is also called **Uniform Memory Access** (UMA). If the memory is on a single node, the memory access time is independent from the node that wants to access the memory.
- **Distributed** if it's **split and each part is located on a node** like in the interconnected network topologies. This memory location policy is also called **Non Uniform Memory Access** (NUMA). In this case, if the modules are placed correctly, we can significantly reduce the memory access. If the memory is divided among nodes, the memory access time depends on the fact that the module on which the requested data is, is directly on the node or on some other node. A node can host multiple modules.

13.2.3 Combination of logical and physical memory location

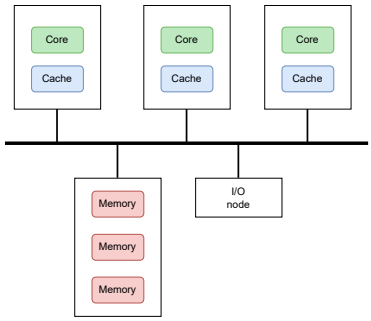
The Memory Address Space model defines logically how memory can be accessed while the physical memory location model defines where each module of memory physically is. These models are however orthogonal, meaning that they can be combined in whatever form. In particular, we obtain

- **Single Logically Shared Centralised (UMA) memories** in which memory is on a single node and can be accessed by all cores (Figure 13.5a).
- **Multiple and Private Centralised (UMA) memories** in which memory is on a single node but each module of memory can be accessed by only one core (Figure 13.5b).
- **Single Logically Shared Distributed (NUMA) memories** in which memory is divided on all nodes but it can be accessed by all cores (a core can even access a memory module not on its node) (Figure 13.5c).
- **Multiple and Private Distributed (NUMA) memories** in which memory is divided on all nodes and each module of memory can be accessed by only one core (Figure 13.5d). Notice that nothing prevents us from putting a module belonging to core $C1$ on core $C2$, however it would be very expensive in terms of memory access time. This means that, usually a module belonging to a core is located on that core's node.

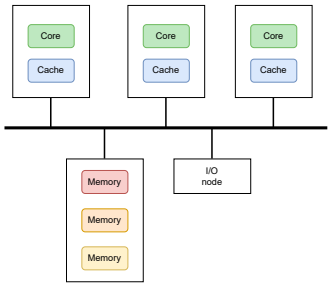
A graphical representation of the architecture listed above is shown in Figure 13.5.

13.3 Cache coherency

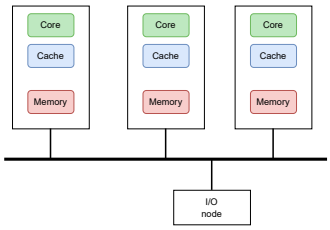
In a multi-core processor, each core has its own low level cache. As always, when talking about caches, we have to remember that there exists different policies to update it. In particular, when executing a write operation, a core can use the write-back or write-through policies (combined with write-allocate or no-write) to update the main memory after a write operation in the cache. This type of coherence is called **vertical coherence** because it involves only a core and the memory.



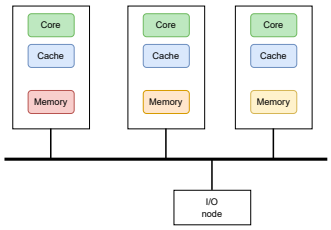
(a) Single Logically Shared Centralised (UMA) architecture.



(b) Multiple and Private Centralised (UMA) architecture.



(c) Single Logically Shared Distributed (NUMA) architecture.



(d) Multiple and Private Distributed (NUMA) architecture.

Figure 13.5: Logical and physical distribution of memory modules. The black line that connects the different nodes is, in general, an interconnection network or a bus.

When discussing multi-core architectures, vertical coherence isn't enough, in fact, we also have to define how caches of different cores interact to keep their values coherent after one of them is written. This type of coherence is called **horizontal coherence** because it involves all the cores. Horizontal coherence is fundamental because, in shared memory architectures, multiple cores might have the same data in their caches (i.e., some shared values might be replicated in multiple caches). To sum it up, in a multi-core architecture we have to define policies and protocols to maintain

- **Vertical coherence** between each node and main memory.
- **Horizontal coherence** among all nodes.

Note that, horizontal coherence is relevant only if the memory is logically shared (independently from its physical location), in fact, otherwise each core can exclusively access only some modules of memory, hence no data would be replicated in different nodes' caches. To handle horizontal coherence, we can define two policies:

- **Write update**. The write update policy imposes that, after a cache write, the update is propagated to all caches.
- **Write invalidate**. The write invalidate policy imposes that, after a cache write by core c, a signal is propagated to all caches, which invalidate (i.e., sets the `valid` bit of a cache block to 0) the block written by c.

Horizontal coherence policies are orthogonal to vertical ones, hence we can have all four combinations of horizontal and vertical polices. Horizontal coherence policies define what type of data should be forwarded (i.e., the updated block or the invalidation signal), however we still need protocols to define how caches should be updated. Cache coherence protocols can be divided in

- **Snooping protocols**.
- **Directory-based protocols**.

13.3.1 Snooping protocols

Snooping protocols monitor the information on the bus (hence they can be used only in bus architectures) and work fine for small systems (2 or 4 cores). In snooping protocols, every cache block has a state, which is shared between all caches that have a copy of that block. When a block is written, the update (or an invalidate signal) is propagated to the whole system, and the cores that have the modified cache block update the state of the modified block. To check if a block is in cache without interfering with the cache itself, we can add an extra read port to the address tag, called for this purposes **snooping tag**, portion of the cache. This port allows to check if the data on the bus is contained in the local cache.

Among all possible combinations of vertical and horizontal cache coherence policies, the most used are:

- **Write through with write update**. This policy is used because data is already on the data bus (since we use the write through policy), hence it's convenient to also propagate the value to the caches.
- **Write back with write invalidate**. Using this policy, we accept the fact that the main memory can be temporarily not coherent, but we can consume less power.

Write through with write update

In the write through with write update, a writing processor broadcasts the new data over the bus and all caches check if they have a copy of the data and, if so, all copies are updated with the new value. This scheme requires the continuous broadcast of writes to shared data. The main advantages of this option are

- The latency between write and read is low.
- The bandwidth is increased.

Write back with write invalidate

When a processor writes a block of cache, it sends an invalidation signal to all other processors. Note that the signal is sent only the first time a block is modified. The writing processor is then free to update the local data until another processor asks for it. This scheme allows multiple readers but only a single writer. This approach deletes all other copies so that there is only one local copy for subsequent writes. The main advantages of this option are that

- We need one transaction per write.
- We exploit spatial locality, since we do one transaction per block.

Modified Shared Invalid

Modified Shared Invalid (**MSI**) is a snooping protocol that uses the **write back with write invalidate** policy. In MSI, a cache block can be in three states

- **Modified** (or dirty). If a block is in the modified state, the cache has the only copy of the block, which is writable, and dirty (i.e., it can't be shared anymore).
- **Shared** (or clean). A clean block (i.e., not modified) can be read and other copies of that block are in other caches.
- **Invalid**. An invalid block contains no valid data.

The memory blocks, on the other hand, can be in one of the following states:

- **Modified** (or dirty). A modified block is valid only in one cache.
- **Shared** (or clean). A shared block is present in all caches and is up to data in main memory.
- **Uncached**. An uncached block is not present in any cache.

Now that we have figured out what are the states in which each block can be, we have to define the transition model, i.e., how a block goes from one state to the other after a certain action. For each state, we will consider all possible actions that the process can do. More precisely, we'll analyse what happens to a single cache entry that contains a cache block, what cache block is loaded in the cache entry and in what state it's loaded. Notice that, in the following analysis, we will call `B0` the block with tag `T0`.

Invalid state If a cache block (containing block with tag `T0`) is in the invalid state, then it can't be read or written, hence every action results in a cache miss. More properly,

- A **read** by cache C_1 on block with tag `T1`, will result is a **read miss** and
  1. A read miss signal is sent on the bus.
  2. If
     - The other caches have tag `T1` in the state `SHARED`, then the block stays shared.
     - Another cache C_2 has tag `T1` in the state `MODIFIED`, then the block is written-back to memory and the block is marked as `SHARED` in C_2.
  3. The block is loaded in cache by C_1 from memory as `SHARED`.
- A **write** by by cache C_1 on tag `T1`, will result is a **write miss**.
  1. A write miss is written on the bus.
  2. If
     - Other caches have tag `T1` in the `SHARED` state, then the block `B0` in other caches goes in the `INVALID` state.
     - Another cache C2 has tag `T1` in the `MODIFIED` state, then block `T1` is written-back to memory and the block in C_2 goes in the `INVALID` state.
  3. Finally, C_1 loads block `T1` from memory, in the `MODIFIED` state.

Shared state When a cache block `B0` is in the `SHARED` state, we can have misses or hits for reads and writes. In particular,

- In case of **read hit** (i.e., the processor wants to read `B0`), the block remains unchanged.
- In case of **write hit** (i.e., the processor wants to write `B0`), cache block `B0` goes into the `MODIFIED` state and a invalidate signal is sent on the bus to invalidate the same block on the other caches.
- In case of **read miss** on block with tag `T1`,
  1. A read miss for block `T1` is placed on the bus.
  2. Block `T1` is taken from memory and cached with state `SHARED`.
- In case of **write miss** on block with tag `T1`,
  1. A write miss for block `T1` is placed on the bus.
  2. Block `T1` is taken from memory, put in cache with state `MODIFIED` and written.

Modified state When a cache block with tag `T0` is in the `MODIFIED` state, we can have misses or hits for reads and writes. In particular,

- In case of **read hit**, the block remains unchanged.
- In case of **write hit**, on block with tag `T0`, block `T0` goes in the `MODIFIED` state but no signal is sent on the bus.
- In case of **read miss** on a block with tag `T1`,

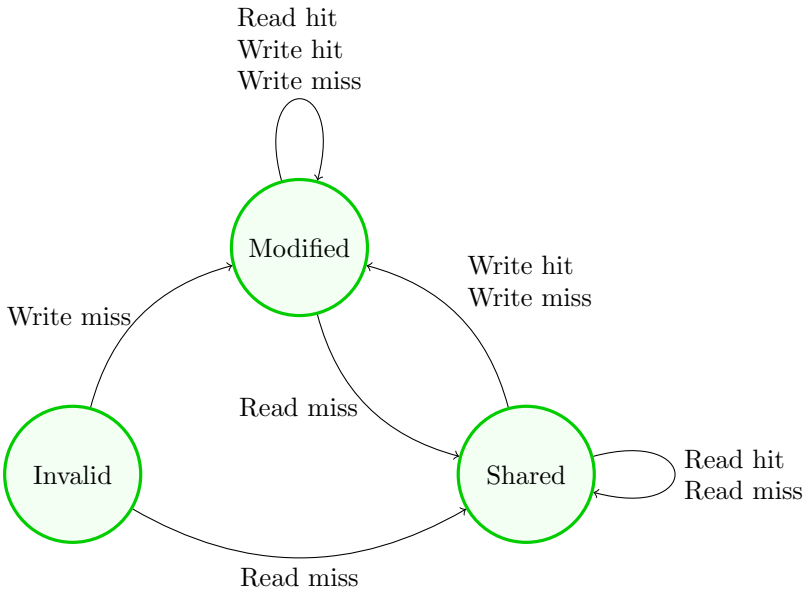


Figure 13.6: State machine of the MSI protocol.

1. The cache block to be replaced is written-back in memory.
2. A read miss signal of block `T1` is placed on the bus.
3. The block `T1` is loaded in cache with state `SHARED`.

- In case of **write miss** on block with tag `T1`,
  1. The cache block to be replaced is written-back in memory.
  2. A write miss signal for block `T1` is placed on the bus.
  3. The block to write is saved in cache with state `MODIFIED` and it's written.

Modified Exclusive Shared Invalid

The MESI protocol is an evolution of the MSI protocol, obtained adding the **exclusive** state to the list of states in which a block can be. Therefore, the states in which a block can be are:

- **Modified** (or dirty). If a block is in the modified state, the cache has the only copy of the block, which is writable, and dirty (i.e., it can't be shared anymore).
- **Exclusive**. A block in the exclusive state is clean and the cache where it is has its only copy.
- **Shared** (or clean). A clean block (i.e., not modified) can be read and other copies of that block are in other caches.
- **Invalidate**. An invalid block contains no valid data.

The main advantage of the MESI protocol is that, when an exclusive block is written, the processor doesn't have to send the invalidation signal on the bus, since no other copy of the block is in other caches.

13.3.2 Directory based protocol

The directory based protocol can be executed on a Single Logically Shared Distributed (NUMA) architecture with an interconnected network. In such architecture, given an address, every processor knows on which node it's placed.

Directory

The directory based protocol is centered around the concept of directory. A directory is a component, located on each node, that stores the coherency state of the node's memory block (i.e., the module). In other words, each node has a directory in which, each entry stores the state of a block of memory of that node. Each entry of the directory contains

- The **state** of the related block. Notice that the state of the memory block is different from the state of the cache. In particular,
  - Each memory block has a state that says in which caches the block is.
  - Each cache block has a state (MSI, Modified Shared Invalid) that says if it has been modified and if it's valid.
- The **processors** that have that same block in the cache (not in memory, because each memory module has different addresses).
- The **owner** of the block, namely, the core that has written the block.

Directory state Each entry of the directory (i.e., each block of memory) can be in three different states:

- **Uncached**. A memory block in the uncached state isn't present in any cache (i.e., in any node). In other words, no node has a copy of that memory block.
- **Shared**. If a memory block is in the shared state, then it is present in one or more caches.
- **Modified**. If a memory block is in the modified state, then only one node has, in its cache, one copy of the block and the block in memory isn't coherent. Basically, a node has modified the block in its cache and the value hasn't been written in memory, yet.

Messages

The directory-based protocol is a **message-oriented protocol**. This means that nodes don't communicate in broadcast (like in snooping protocols) but they exchange precise unicast messages (because they can exploit the interconnection network). Each message sent has to be acknowledged, too. This characteristic makes the protocol **very scalable**.

Before describing the type of messages that nodes can exchange, let us classify the nodes. In particular, a set of messages needed to execute an operation is called **transaction**. For instance, after a read or a write miss a node has to ask the value to the main memory. This operation is done with a transaction. In each transaction we can recognise

- A **local node**. The local node is the node that requests a block from memory.
- An **home node**. The home node is the node that has the memory location needed by the local node.

- A **remote node**. The remote node is the node that has a dirty (i.e., modified) or shared copy of a block in cache.

In a transaction,

- The local and home node can coincide.
- The remote and home node can coincide.
- The remote and local home can't coincide by definition because if the local node were also the remote node, it would have had the block in the cache, hence it wouldn't need to ask the block to the home node.

Now that we have classified each node in a transaction, let us define what messages can nodes exchange.

Read miss A **read miss** message is sent from the local node to the home node to notify a read miss and contains the identifier `P` of the processor in the local node and the memory address `ADDR` requested.

TYPE	SENDER	RECEIVER	PAYLOAD
Read miss	Local cache	Home directory	P, ADDR

This message is used by the local node to ask the block at address `ADDR` in the home node and make processor `P` a shared reader. Basically, when the home nodes receives a read miss message, it

1. Sends the requested block to the local node.
2. Adds processor `P` to the list of sharers in the sharer bits of the home directory.

Write miss A **write miss** message is sent from the local node to the home node to notify a write miss and contains the identifier `P` of the processor in the local node and the memory address `ADDR` requested.

TYPE	SENDER	RECEIVER	PAYLOAD
Write miss	Local cache	Home directory	P, ADDR

When the home node receives a write-miss message, it

1. Sends the requested data (i.e., the block at address `ADDR`) to `P`.
2. Changes to `MODIFIED` the state of the block sent to `P` in the home directory.
3. Sends an invalidate message to the other caches (more on this message later) to tell them that the only valid value is on `P`'s cache.

Data Value Reply A **Data Value Reply** message is sent by the home directory, to the local directory, and contains the data value requested by the local directory with a read or write miss.

TYPE	SENDER	RECEIVER	PAYLOAD
Data Value Reply	Home directory	Local cache	Data@ADDR

Home invalidate An **home invalidate** message is sent by the local cache to the home directory and contains the address `ADDR` to invalidate. This message is used by the local node to ask to the home to send an invalidate message to all remote caches that are caching the block at address `ADDR`.

TYPE	SENDER	RECEIVER	PAYLOAD
Invalidate	Local cache	Home directory	ADDR

Remote invalidate A **remote invalidate** message is sent by the home node to the remote nodes and contains the address `ADDR` to invalidate. This message is used by the home node, after receiving a Home Invalidate message, to invalidate a shared copy of the block with address `ADDR` in all remote caches.

TYPE	SENDER	RECEIVER	PAYLOAD
Invalidate	Home directory	Remote cache	ADDR

Fetch A **fetch** message is sent by the home directory to the remote cache that owns a block at address `ADDR`.

TYPE	SENDER	RECEIVER	PAYLOAD
Fetch	Home directory	Remote cache	ADDR

A fetch message is used to get the block in the home memory at address `ADDR` from the remote node that owns it. In particular,

1. The home directory asks the block at address `ADDR` to the remote owner with a **fetch** message.
2. The owner sends the requested data to the home directory through a Data Write-Back message (more on that later).
3. The remote cache changes the cache block state from `MODIFIED` to `SHARED`.
4. The block state of block at `ADDR` in the home directory changes from `MODIFIED` to `SHARED`.

Fetch and invalidate A **fetch and invalidate** message is sent by the home directory to the remote cache that owns a block at address `ADDR`.

TYPE	SENDER	RECEIVER	PAYLOAD
Fetch invalidate	Home directory	Remote cache	ADDR

A fetch message is used to get the block in the home memory at address `ADDR` from the remote node that owns it and invalidate it in the remote cache. In particular,

1. The home directory asks the block at address `ADDR` to the remote owner with a **fetch and invalidate** message.
2. The owner sends the requested data to the home directory through a Data Write-Back message (more on that later).
3. The remote cache changes the cache block state from `MODIFIED` to `INVALID`.
4. The block state of block at `ADDR` in the home directory doesn't change but the home changes the owner bits.

Basically, we are changing the owner of the block at address `ADDR`. This type of message is used after a write miss on a block not owned by the home directory. The idea is that

1. The local node asks to write a block to the home.
2. The home node finds out that the block is not shared but it's owned by some other node.
3. The home node requests the block to the remote node and tells it that it isn't the owner anymore.
4. The home node sends the block just fetched to the local node and makes it the new owner.

Data write-back A **data write-back** message is sent by the remote cache to the home directory and contains the data requested by the memory.

TYPE	SENDER	RECEIVER	PAYLOAD
Data write-back	Remote cache	Home directory	ADDR, Data@ADDR

A data write-back message is used to write-back a data value from the remote cache (which was dirty) owner to the home memory.

State machine

After understanding what messages can be exchanged, let us check how a directory entry can change state. To analyse the state machine we will consider a chip with 4 CPUs (identified by `P0`, `P1`, `P2`, `P3`).

Uncached state Say a block `B0` is in the uncached state and a read miss message from processor `P1` is received. In this case,

1. Processor `P1` is added to the list of sharers.
2. The block goes in the `SHARED` state.
3. A Data Value Reply with the content of block `B0` is sent to `P1`.

On the other hand, if a write miss message (together with an invalidate message) from `P1` is received,

1. Processor `P1` is added to the list of sharers.
2. The block goes in the `MODIFIED` state.
3. A Data Value Reply with the content of block `B0` is sent to `P1`.

Shared state Say a block `B0` is in the shared state on processor `P0`. When `P0` receives a read miss from the local cache of node `N2`,

1. The requested data are sent with a Data Value Reply to the local cache on `N2` and
2. `N2` is added to the sharer bits.
3. The state of the block `B0` is not modified and remains `SHARED`.

When P0 receives a write miss (together with an invalidate message) from the local cache of node N2,

1. The requested data are sent with a Data Value Reply from the home memory on N0 to the local cache of node N2.
2. Invalidate messages are sent from the home node N0 to the remote sharers.
3. The sharers bits are set to 0, except for the bit of N2 (i.e., the local node).
4. The state of the block becomes MODIFIED.

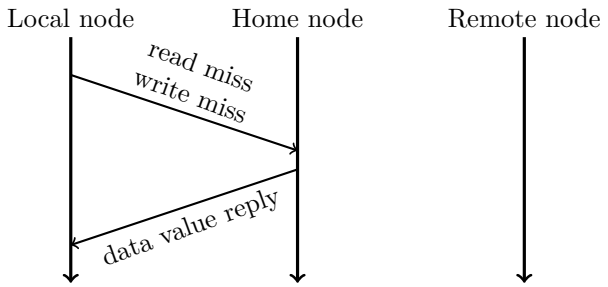


Figure 13.7: Read miss transaction on a shared or uncached block or a write miss transaction on an uncached block.

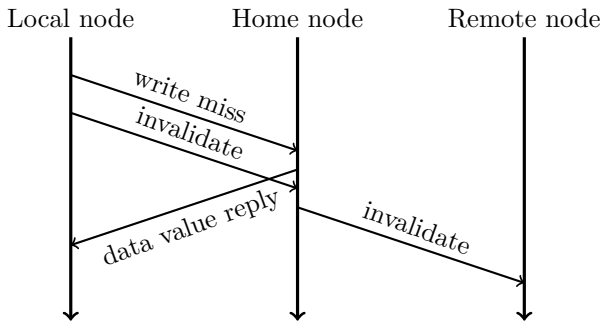


Figure 13.8: Write miss transaction on a shared block.

Modified state Say a block B0 is in the modified state on processor P0 and the current node of B0 is on node N3, which is the owner of B0. When P0 receives a read miss from the local cache of node N2,

1. The home node N0 sends a fetch message to the owner node N3, causing state of the block in the owner's cache to transition to SHARED.
2. The owner N3 sends the data to the home directory on N0, through a Data Write Back message. Upon receiving the data, N0 updates the memory.

3. The data that has been fetched are sent to the local cache on `N2` with a Data Value Reply message.
4. The identity of the requesting processor `P2` is added to the sharers bits. The identity of the owner `P3` isn't removed from the shared bits since it still has a readable copy.
5. The block state in `N0`'s directory is set to `SHARED`.

When `P0` receives a write miss message (together with an invalidate message) from a node `N2`,

1. A fetch and invalidate message is sent to the old owner `N3`.
2. The cache block on `N3` goes in the `INVALID` state.
3. The old owner `N3` sends the block `B0` to the home directory `N0` with a Data Write Back message.
4. Block `B0` is sent to the requesting node `N2` with a Data Value Reply.
5. Processor `P2` is set as the new owner on `N0`'s directory.
6. The sharers bits are set to 0, except for the bit of `P2` (i.e., the local node), which is the new owner.
7. The state of the block remains `MODIFIED`, but owner changed.

Finally, when `P0` receives a data write-back from a remote owner `N2`, the owner's cache is replacing the block and therefore `N3` must write it back to home. This makes the memory copy up to date (the home directory becomes the owner) and the block becomes `UNCACHED`, and the sharer bits are set to an undefined value.

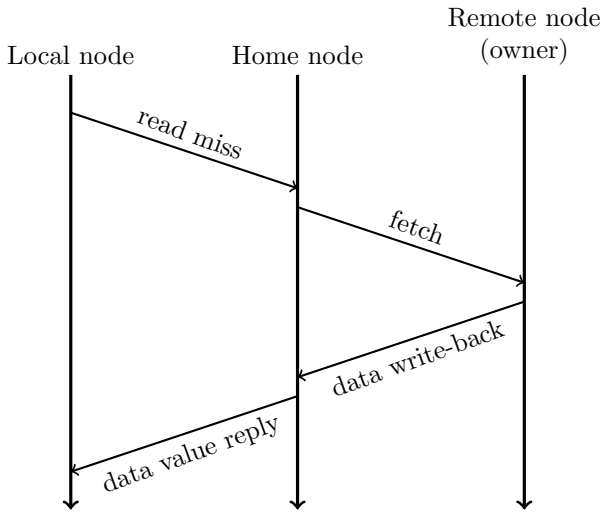


Figure 13.9: Read miss transaction on a modified block.

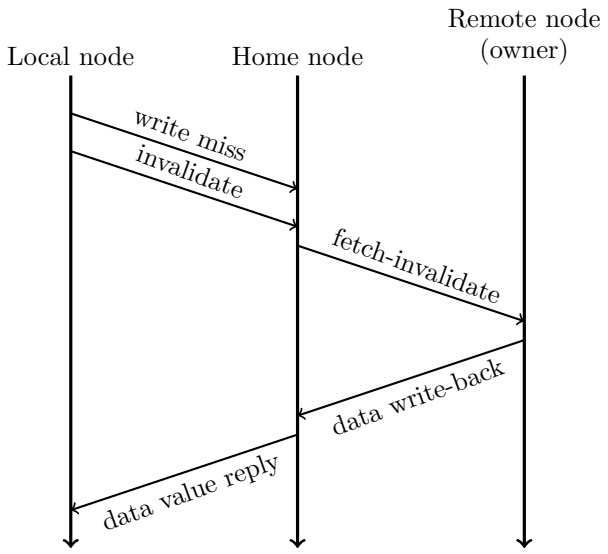


Figure 13.10: Write miss transaction on a modified block.

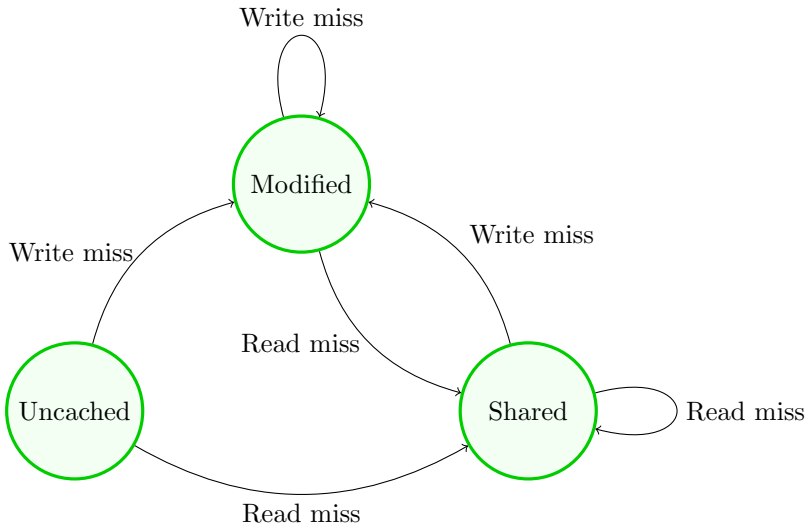


Figure 13.11: State machine of a directory block for the directory based protocol.

Chapter 14

Data level parallelism

14.1 Introduction

Instruction level parallelism can increase the throughput of a processor, however only up a certain point. To further increase the throughput of a core, we can also exploit data level parallelism and execute the same instruction on multiple, independent data. Consider for instance a loop as the one below

```
for (int i=0; i<N; i++) {
    v[i] = v[i] + a;
}
```

As we can see, the value of `v` in position `i` doesn't depend on the values in the other positions, hence we could execute the operation `v[i] = v[i] + a` for different values of `i` in parallel. Considering the example of loops, we can exploit data level parallelism if the loop doesn't have loop-carried dependencies. In general an instruction `instr res op1 op2` can be executed in parallel on multiple values of `op1` if all the values of `op1` are independent from the other values of `op2` (the same is true for `op2`). In our example, all the elements of `v` have to be independent, or better said, for each index `i`, `v[i]` has to be independent from the other values `v[j]`.

14.1.1 Taxonomy

The systems that exploit data level parallelism can be classified in

- **Single Instruction Single Data (SISD)**. SISD processors are simple scalar processor that don't exploit data level parallelism. Processors of this type have
  - A **single instruction stream** (i.e., one program counter).
  - A **single data stream**. This means that an instruction is executed on a single datum.
- **Single Instruction Multiple Data (SIMD)**. SIMD processors,
  - Use a **single instruction stream**, i.e., a single program counter.
  - Execute an instruction on **multiple data**.
- **Multiple Instructions Single Data (MISD)**. MISD processors are not practical, hence we have no practical examples of processors of this type.

- **Multiple Instructions Multiple Data (MIMD)**. MIMD processors
  - Handle **multiple instruction streams**, i.e., multiple program counters, hence multi-threading.
  - Apply each instruction of the instruction stream on **multiple data**.

14.2 Single Instruction Multiple Data processors

Among all types of processors that exploit data level parallelism, SIMD processors are of great interest because

- They are **energy efficient** (i.e., consume few energy), especially in multi-core architectures.
- We can **think in a sequential way**, in fact, having a single program counter, instructions are executed one after the other.

14.2.1 Architecture

The main components of a SIMD processor are

- A **controller**. The controller issues instructions and data.
- A **set of processing elements** each of which is connected to a partition of the memory. Basically, memory is partitioned and each element has visibility only on a partition.
- An **interconnection network**. The interconnection network connects the processing elements and the controller.

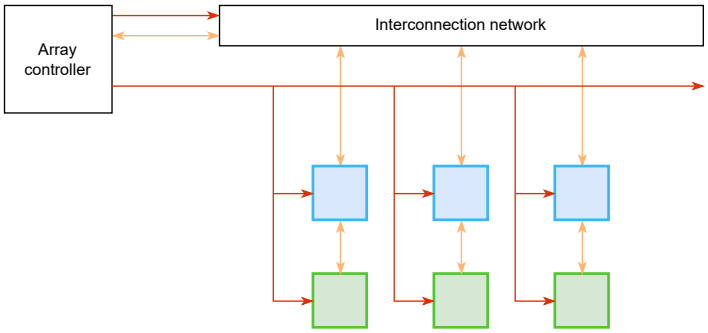


Figure 14.1: A SIMD architecture. Data is moved on the red bus while control signals are sent on the orange wire. Memory is drawn in green while the processing elements are coloured in blue.

14.2.2 Variations of the SIMD architecture

SIMD architectures can be further divided into

- **Vector architectures**.
- **SIMD extensions**.
- **Graphics Processor Units** (GPUs).

14.3 SIMD vector architectures

SIMD vector architecture use vector register files to execute an instruction on multiple data. This means that we need a new set of instructions (or we simply have to expand the already present ISA) that includes the instructions to operate on vector registers.

14.3.1 Cray1

Cray1 is a SIMD vector implementation. Cray1 processors

- Use a **load/store architecture** (i.e., data has to be stored in the registers to be used and operations can't be done directly specifying memory locations).
- Use a **vector register file**.
- Use a **scalar register file**.
- Have **highly pipelined functional units**.
- Use an **interleaved memory system**.
- Don't use **caches**.
- Don't use **virtual memory**.

Architecture

A Cray1 processor is made of

- A **vector register file**. The vector register file contains multiple vector registers, each of which contains multiple **elements**. Each element has a fixed bit length. The vector register file should have at least two read ports and one write port. In other words, a register is a vector of elements and each element has a fixed length.
- **Vector functional units**. The vector FUs are usually fulled pipelined.
- **Vector load-store units**. The vector load-store units are fully pipelined and are used to load data from memory into vector registers.
- A **scalar register file**. The scalar register file contains scalar values. The registers can also be divided in integer and floating point registers.
- A **cross-bar network**. The interconnection network the functional units, the load-store units and the register files.

Vector instructions

Since Cray1 uses vector registers, we have to introduce new instructions to handle vector operations. An instruction that takes as input two vectors is usually written appending `vv` to the name of the equivalent scalar instruction. An instruction between a vector and a scalar is written appending `vs`.

14.4 VMIPS

VMIPS is an implementation of the SIMD vector architecture built on top of MIPS.

14.4.1 Registers

The vector register file of a VMIPS processor is made of 8 vector registers, each made of 64 elements, each of 64-bits. This means that each register is $64 \cdot 64 = 4096$ bits long. The register file has at least 16 read ports and 8 write ports.

Moreover, as a normal MIPS processor, a VMIPS processor has 32 integer registers and 32 floating point registers.

14.4.2 Chaining

VMIPS processors use a technique called chaining to execute instructions in parallel. Chaining can be applied in different ways. To help understand how chaining works, we can consider the following example, which represents a DAXPY instruction.

```
1    for(i=0; i<64; i++){
2        Y[i] = a * X[i] + Y[i]
3    }
```

Listing 14.1: DAXPY C code.

The aforementioned piece of code (Listing 14.1) can be written using vector instructions as follows

```
1    ld f0, a
2    lv v1, Rx
3    mulvs v2, v1, f0
4    lv v3, Ry
5    addvv v4, v2, v3
6    sv Ry, v4
```

Listing 14.2: DAXPY assembly code.

Notice that the loop has been translated in a single iteration because the vector registers can hold 64 values and the loop needs to iterate over 64 values (i.e., all the 64 values of `Y` and `X` can be loaded in one register each).

Executing the instructions above truly in parallel is practically impossible. To understand why, consider for instance `addvv v4, v2, v3`. Executing this instruction in parallel on all cells of `v2` and `v3` is practically impossible because

- We would need a bus with a lot of bandwidth since the vector registers are 4096 bits long.
- We would need a lot of functional units, 64 to be precise, to execute each of the 64 instructions `v4[i] = v2[i] + v3[i]`.

Having ruled out the possibility to fully execute the instructions in parallel, we have to introduce chaining to execute instructions on multiple data, without using too many resources. Since applying an instruction on all the data is unfeasible, we have to start applying the instruction only on some data and use multiple clock cycles to apply the instruction on the whole vector. This means increasing the duration of a vector instruction, since, for each clock cycle, we apply the instruction only on a portion of the vector register.

No chaining

The first method to executes instructions on multiple data is called no chaining and doesn't actually exploits data level parallelism. This technique allows the processor to execute the vector instructions one after the other. In other words, the processor starts executing a vector instruction only after the previous one finished its execution. Notice that, since every instruction is executed after the previous one, a VMIPS processor with no chaining needs only one functional unit of each type.

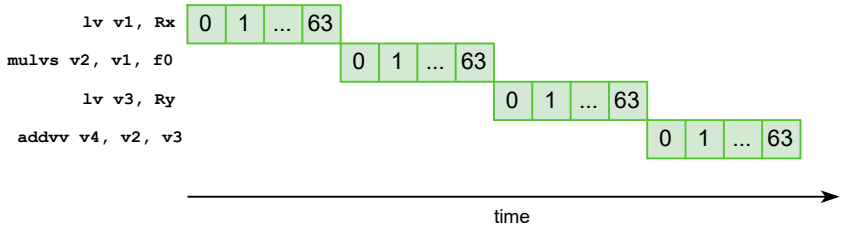


Figure 14.2: The execution of the DAXPY instruction with no chaining. The numbers inside the blocks represent the indices of the vector register.

Chaining

Chaining improves the performance of no-chaining using more resources. In particular, each instruction can start as soon as the individual elements of its vector source operand become available. Consider for instance the dependency between instruction 2 and 3 in Listing 14.2. Instruction 3 can start as soon as `v1[0]` is ready, hence at the clock cycle after the one in which instruction 2 is issued. Basically, we are extending the concept of forwarding to registers. Notice that in this case, one functional unit for each type might not be enough, in fact, since each operation takes 64 clock cycles, the loads at instruction 2 and 4 will for sure be executed in the same clock cycle, hence we need at least 2 load-store units (if we consider the store at instruction 6, which is executed in parallel, the number of store units becomes 3).

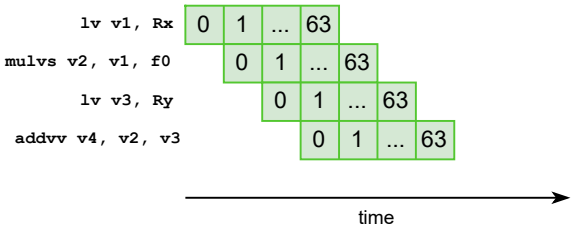


Figure 14.3: The execution of the DAXPY instruction with chaining. The numbers inside the blocks represent the indices of the vector register.

Convoys chaining

An intermediate solution between chaining an no chaining uses convoys to issue instructions in batches. This solution requires only one functional unit per type. In particular, a processor can issue an instruction as soon as

- The functional unit needed is available.
- The individual elements of its vector source operand become available.
- The previous operation has started.

This means that instructions are issued in a chaining fashion until one instruction needs a functional units already occupied. The set of instructions issued in subsequent clock cycles is called convoy. Using convoys we were able to exploit data parallelism with a limited number of resources. The time needed to execute a convoy is called **chime** and can be used to compute the clock cycles needed to execute a set of instructions

$$N_{clock\ cycles} = N_{convoys} \cdot T_{chime}$$

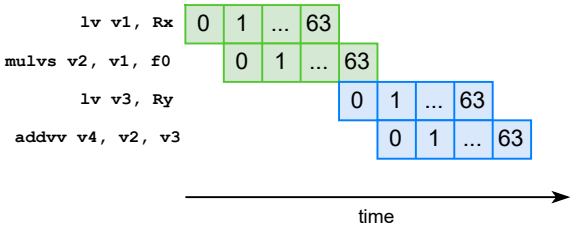


Figure 14.4: The execution of the DAXPY instruction with convoy chaining. The different convoys are represented with two different colours. The numbers inside the blocks represent the indices of the vector register.

14.4.3 Lanes

To further reduce the number of clock cycles needed to execute a piece of code, we can try and execute some arithmetic instructions in parallel on a single functional unit. To achieve this goal we have to introduce a new type of ALU that can handle multiple instructions in parallel, called lanes.

14.4.4 Variable size vectors

The example we used to describe the VMIPS architecture used vectors of 64 element, which is the same number of elements in a vector register. However, some programs might use vectors that are smaller or bigger than the number of elements in a vector register. In particular,

- If the length of the vector used in the program is smaller than the vector register, then we have to use a special register, called `vector_length` that controls the length of a vector operand (i.e., register).
- If the length of the vector used in the program is bigger than the one of the vector register, then we have to split the program's vector in multiple vector registers and use multiple steps to compute the operation. This solution is called **strip mining**.

14.4.5 Conditional loops

Another interesting aspect to analyse is how data level parallelism can be applied to loops in which an operation has to be executed only if a certain condition is true (see code above).

```
for(i=0; i<10; i++){
    if (i%2==0){
        v[i] = v[i] + 2;
    }
}
```

To handle these cases, a VMIPS processor uses a special vector register called `mask`. The `mask` vector behaves like a mask that tells in which elements of the result vector to write the result. In other words, the `mask` vector register blocks the instruction from writing the result on the elements of the vector that don't satisfy the condition.

14.4.6 Stride loading

Usually, a data is loaded in vector registers from contiguous addresses of memory. Basically, if we want to load the first 64 elements of a vector in a vector register, we load the first element of the vector in the first element of the register, the second element of the vector (which is immediately after the first in memory) in the second element of the register and so on. However, when we consider matrices, one might want to do some operations column by column (but the matrix is saved by rows in memory). To load the elements in the columns (i.e., in non adjacent blocks of memory) a VMIPS processor can use the `lvws` instruction that loads in register `v1` the values separated by `R2` addresses, starting from address `R1`.

```
lvms v1, (R1, R2)
```

The same operation could be written in code as

```
for (i=0; i<REG_LEN; i++){
    v1[i] = MEMORY[R1 + i*R2]
}
```

14.4.7 Gather-scatter operators

Many applications in which vector processors are widely used works with sparse matrices. To optimise the operations on these types of matrices VMIPS offers two special instructions

- The `gather` instruction allows to collect only the useful values of a sparse matrix and save them in a register. Basically, we are going from a sparse matrix to a compact form.
- The `scatter` instruction allows to distribute the values in a vector register (previously saved with the `gather` instruction) in memory. Basically we are going from a compact form to a sparse matrix.

Chapter 15

Graphical Processing Units

15.1 Introduction

Graphical Processing Units (GPUs) are accelerators for graphics elaboration, i.e., the translation of a 3D model in a 2D image. Even if the first usage of GPUs has been for graphics, GPUs have also evolved to execute general purpose computations in which instructions can be executed on different data independently. In general, a GPU is used for tasks that

- Involve mapping operations (e.g., from 3D space to 2D space).
- Require few synchronisation.
- Execute multiple coarse-grained functions that can be executed in parallel and at the same time on different data.

Before analysing the architecture of a GPU in detail, let us say that a GPU usually has its own memory (called graphical or video memory).

The main goal of a GPU is to its maximise the throughput, differently from the goal of a CPU which tends to minimise the instruction latency.

15.2 Architectural overview

GPUs are connected to the CPU using a PCIe bus. The connection between CPU and GPU is one of the bottlenecks of the system, hence manufacturers have developed custom interconnection busses to increase the bandwidth of this link.

15.2.1 Rendering

To describe the structure of a GPU, we should understand what GPUs have initially been designed for. GPUs were initially (and are still now) used for rendering, i.e., translating a 3D model in a 2D image to show on the screen. The translation process is called **graphics pipeline** and basically returns the colour each pixel of the output image. A graphics pipeline is basically a sequence of functions (written in custom languages like OpenCL, an extension of C), called **shaders**, applied on a single fragment. A fragment is the basic working unit that will eventually result in a pixel. Since pixels are independent one another, an operation on a fragment is independent from the other

fragments (it only depends on the 3D model). Usually, shaders contain a lot of matrix and vector multiplications or sine and cosine operations but rarely they contain conditional statements or loops.

15.2.2 Many-core architecture

Multi-core

Now that we know what process a GPU should execute, let's try to build one incrementally, starting from a normal CPU. A normal CPU is made of

- One or more **functional units**.
- A **fetch and decode module** to fetch instructions form memory.
- A **cache**.
- An **optimisation control logic**. This circuitry is used to enforce instruction level parallelism and to manage caches.

However, the enhancements brought by instruction level parallelism (e.g., dynamic scheduling) have little effect on a graphics pipeline. This mean that we can get rid of the optimisation control logic that handles instruction level parallelism and of the cache (the reason will be clear later on). This leads us to a new CPU with only

- One or more **functional units**.
- A **fetch and decode module** to fetch instructions form memory.

This new CPU has slightly worst performance with respect to the enhanced version, however it's radically smaller (since the optimisation logic occupies a lot of space). This means that we can use the spared space to duplicate the small CPU we have obtained. Moreover, if we remember that we are building a processor to execute data-parallel instructions, we can also reduce at a minimum the synchronisation and coherency logic between the two cores because the cores don't have to work on the same data but simply execute the same operations on different data. If we keep adding simple cores to the architecture described above, we obtain a **multi-core architecture** that can heavily parallelise execution on independent data. Each core in a multi core architecture is usually called **unit**.

Let's stop for a moment to highlight an important thing. Given that a core is simpler than the one we initially considered (i.e., the enhanced one), it is slower that the original CPU. However, if we consider the architecture as a whole (i.e. the performance of all the cores that execute a single program), we get much better performance and throughput.

Wrap architecture

Let us now focus on the architecture of a single core of the multi-core architecture. Since the main point of the graphics pipeline is to execute shaders in parallel on different data, we would like to apply data-level parallelism also to a single core. This means that we can use SIMD vector processors instead of simple SISD processors to implement every core. The SIMD cores used in GPUs are however slightly different from those analysed before. In particular, initially we said that to work in parallel on different data we needed a new ISA with vector-dedicated instructions (e.g. `addvv`). SIMD cores used in GPUs use a different approach, namely, the program to execute is

compiled using the basic scalar ISA but the fetch and decode unit is modified. More precisely, the fetch and decode unit is able to execute the same instruction on multiple context using threads. Basically, when the core has to execute an instruction in parallel, it can use a set of threads (also called streams), called **warp** (i.e., a warp is a set of threads) that share the same PC and execute the same instructions but on different data.

Many-core architecture

If we put together multi-core and warp-cores we obtain the so called **many-core architecture**,i.e. , a multi-core architecture in which each core is a SIMD vector core that uses warps. The many-core architecture exploits two levels of parallelism

- One brought by the **multi-core component**.
- One brought by the **SIMD processors**.

15.2.3 Conditional statements

Initially, we assumed that shaders have very few conditional statements, however, it's still possible to have some. This is a problem because every thread of a warp should execute the same instructions (remember that we use a program counter for each warp, hence for all threads of a warp), however a condition might be true on the data used by thread T_1 and false on the data of T_2. This means that T_1 and T_2 should execute different instructions, but that's impossible because they have a single shared program counter. One way to solve this problem is to execute all the possible paths sequentially (e.g., in an if clause execute the `then` branch, followed by the `else` branch) and activate a thread only when the program counter is on the instruction that the thread should execute. For instance, if a condition is true for thread T_1 but false for thread T_2, we move the PC on the `then` and `else` branches, however T_1 will be activated only when the program counter is in the `then` branch and T_2 only when the core's executing the `else` branch. Notice that, this way of handling conditions decreases performance but it inevitable if we want a single program counter for warp. An important thing to point out is that the process described above is applied on a single warp of a SIMD core. Say for instance that all threads on SIMD_1 say that a condition is true while all threads on SIMD_2 say that the same condition is false. In such situation we don't have to execute both the branches of the if clause because we are evaluating two different cores that have different program counters.

15.2.4 Memory access

GPUs usually have their own memory called graphics (or video) memory. Moreover, differently from CPUs, GPUs don't have (or at least didn't have in their earlier stages) caches. The reason for this choice is that handling caches (i.e., cache coherency), especially in processors with many cores (even hundred of cores), is very expensive and requires some hardware that occupies a lot of space. However, memory access is much slower (around 100 times slower) that instruction execution. To solve this problem we can use a technique called **warp interleaving**. First of all let's say that we are focusing on a single SIMD core. The basic idea behind warp interleaving is to use multiple warps, each with its program counter, on a SIMD core. Before explaining how this technique works, let us clarify a bit the context. We have, on a single core multiple warps each made of multiple threads. Each warp has a program counter which is shared among all threads of that warp. Note that, it's possible to have multiple PCs because the operations executed on a warp don't depend on those executed on the other warp. Hoping that the context is clear, let us analyse how does warp

interleaving work. The idea is that, a SIMD core starts executing a warp until it needs to access the memory. When memory access is required, the warp is stopped and another warp starts execution (and in the meanwhile data is sent from memory to the core). Basically we are hiding memory stalls with warps. This technique is coherent with the fact that GPUs try to maximise throughput, in fact in this case we are trying to maximise the usage of a resource (i.e. the core) to increase throughput, yet decreasing instruction latency because a warp shares a resource with other warps, hence it needs more time to complete.

Caches

In recent years, GPU designers started adding caches inside GPUs. One example is to use a L1 cache for each core and a L2 shared cache that is directly connected to memory. One important thing to understand is that, when we add caches to GPUs we can't control their coherency because it would be too expensive. This isn't however a big problem since threads access different data.

Read only memory

Graphic memories in GPUs use also read-only memories to store shaders.

Block access

Since many cores have to access memory concurrently, we can divide a video memory in many blocks so that each core can access a different block, hence multiple cores can access memory at the same time and the throughput increases.

Appendix A

Performance

In this chapter we will give some general definitions to compute the performance of processors and memories.

A.1 Processors

A.1.1 CPU time

The CPU time t_{CPU} (also execution time) is the time needed by a CPU with frequency f_{CPU} to execute N_{cc} clock cycles.

$$t_{CPU} = N_{cc} \cdot f_{CPU} = N_{cc} \cdot \frac{1}{T_{clock}} \tag{A.1}$$

To increase the execution time of a CPU we can

- Decrease the number of clock cycles needed to execute a program.
- Increase the frequency of the processor.
- Decrease the length of a clock cycle T_{clock}.

The execution time can also be computed as

$$t_{CPU} = IC \cdot CPI \cdot T_{clock} = \frac{IC \cdot CPI}{f_{CPU}} \tag{A.2}$$

where

- IC is the Instruction Counter, i.e., the number of instructions of the program.
- CPI it the Clock Per Instruction, i.e., the number of clock cycles needed, on average, to execute an instruction.

Differentiating instructions

In a program, some types of instructions are more frequent than others. If we want to obtain a more precise execution time we can consider the CPI for each different type of instruction. In particular,

we can rewrite the number of clock cycles in A.1 with

$$N_{cc} = \sum_{i=0}^{N} CPI_i \cdot f_i$$

where

- N is the number of different types of instructions.
- CPI_i is the average number of clock cycles needed to execute an instruction of type i.
- f_i is the frequency with which an instruction of type i is present in the program. This value can be computed as the ratio between the number of instruction IC_i of type i and the total number of instructions IC.
$$f_i = \frac{IC_i}{IC}$$

Given these considerations, the CPU time can be rewritten as

$$t_{CPU} = \sum_{i=0}^{N} CPI_i \cdot f_i \cdot \frac{1}{T_{clock}} \tag{A.3}$$

A.1.2 Clock Per Instruction and Instructions Per Clock

The CPI is the number of clock cycles needed, on average, to execute an instruction. The IPC is the number of instructions executed, on average, in a clock cycle. These two numbers are related by the following formula

$$CPI = \frac{1}{IPC} \tag{A.4}$$

A.1.3 Millions of Instructions Per Second

The Million of Instructions Per Second $MIPS$ metric measures how many millions of instructions are executed every second (don't say!). The $MIPS$ can be computed as

$$MIPS = \frac{IC}{t_{CPU} \cdot 10^6} = \frac{IC}{\frac{IC \cdot CPI}{f_{CPU}} \cdot 10^6} = \frac{f_{CPU}}{CPI \cdot 10^6} \tag{A.5}$$

A.1.4 Amdahl's law

Amdahl's law allows to compute how faster a processor executes a program when a new feature is applied to it. In other words, we are computing the speed up of the enhanced processor with respect to the basic version. The speedup $S(E)$ given a feature (or enhancement) E can be computed as the ratio between the execution time without E and with E.

$$S(E) = \frac{t_{CPU,without}(E)}{t_{CPU,with}(E)} \tag{A.6}$$

Equivalently, we can use a performance metric (e.g., MIPS), but in this case we have to invert numerator and denominator.

$$S(E) = \frac{MIPS_{with}(E)}{MIPS_{without}(E)} \tag{A.7}$$

Features on fractions of the processor

In some cases, a feature can improve performance only on a fraction of the system and has effect only on some tasks. To modify the formula for the speedup we can rewrite the execution time $t_{CPU,with}(E)$ with the feature E as

$$t_{CPU,with}(E) = (1 - F + \frac{F}{S}) \cdot t_{CPU,without}(E)$$

where

- F is the fraction of the computation time in the original machine that can take advantage of the feature.
- S is the speedup gained by the components affected by the feature.

If we replace this formula in the formula for the speedup we obtain

$$S(E) = \frac{t_{CPU,without}(E)}{t_{CPU,with}(E)} \tag{A.8}$$

$$= \frac{t_{CPU,without}(E)}{(1 - F + \frac{F}{S}) \cdot t_{CPU,without}(E)} \tag{A.9}$$

$$= \frac{1}{1 - F + \frac{F}{S}} \tag{A.10}$$

A.2 Pipelined processors

A.2.1 Clock cycles

In a pipeline, the number of clock cycles needed to execute a program is the sum of

- The number of instructions IC.
- The number of stalls N_{stalls}.
- 4 clock cycles because the last instruction ends executing 4 clock cycles after being fetched (the pipeline has 5 stages).

$$N_{cc} = IC + N_{stalls} + 4 \tag{A.11}$$

A.2.2 Clocks Per Instructions

The number of clock cycles per instruction CPI can be rewritten, considering the new formula for the number of clock cycles, as

$$CPI = \frac{N_{cc}}{IC} = \frac{IC + N_{stalls} + 4}{IC} \tag{A.12}$$

Average CPI

The average CPI is higher than the ideal CPI (which is 1) because of stalls in the pipeline.

$$CPI_{avg} = 1 + SPI \tag{A.13}$$

where SPI is the Stall Per Instruction, i.e., the average number of stalls per instructions

$$SPI = \frac{N_{stall,tot}}{IC} = \frac{N_{stall,struct} + N_{stall,data} + N_{stall,control} + N_{stall,memory}}{IC} \tag{A.14}$$

A.2.3 Loop performance

Given a n iterations of a loop containing m instructions that generate k stalls per iteration,

- $$IC_{iter} = m \tag{A.15}$$
- $$N_{cc,iter} = IC_{iter} + N_{stalls,iter} + 4 \tag{A.16}$$
- $$CPI_{iter} = \frac{IC_{iter} + N_{stalls,iter} + 4}{IC_{iter}} \tag{A.17}$$
- $$MIPS_{iter} = \frac{f_{CPU}}{CPI_{iter} \cdot 10^6} \tag{A.18}$$

A.2.4 Asymptotic loop performance

Given an infinite number of iterations ($n \to \infty$) of a loop containing m instructions that generate k stalls per iteration,

- $$IC_{asymptotic} = m \cdot n \tag{A.19}$$
- $$N_{cc,asymptotic} = IC_{asymptotic} + N_{stalls,asymptotic} + 4 \tag{A.20}$$
- $$CPI_{asymptotic} = \lim_{n\to\infty} \frac{IC_{asymptotic} + N_{stalls,asymptotic} + 4}{IC_{asymptotic}} \tag{A.21}$$
$$= \lim_{n\to\infty} \frac{m \cdot n + k \cdot n + 4}{m \cdot n} \tag{A.22}$$
$$= \frac{m + k}{m} \tag{A.23}$$
- $$MIPS_{iter} = \frac{f_{CPU}}{CPI_{asymptotic} \cdot 10^6} \tag{A.24}$$

A.3 Memory

A.3.1 Cache hit rate

The hit rate f_{hit} is frequency at which the processor finds the block it's looking for in the cache and it's computed as the ratio between the number of cache hits $N_{cache\ hits}$ and the number of memory accesses $N_{memory\ accesses}$.

$$f_{hit} = \frac{N_{cache\ hits}}{N_{memory\ accesses}}$$

A.3.2 Cache hit time

The hit time t_{hit} is the time required to access a block of data in the upper level cache following a cache hit. t_{hit} can be expressed in number of clock cycles or in seconds (a common value is 1 ns).

A.3.3 Cache miss rate

The miss rate f_{miss} is the frequency at which the processor doesn't find a block in the cache and it's computed as the ratio between the number of cache misses $N_{cache\ misses}$ and the number of memory accesses $N_{memory\ accesses}$.

$$f_{miss} = \frac{N_{cache\ misses}}{N_{memory\ accesses}}$$

A.3.4 Cache miss penalty

The miss penalty $t_{penalty}$ is the time needed to access the lower level of cache and replace the block in the upper level of cache. Usually the miss penalty is much larger than the hit time (i.e., $t_{hit} \ll t_{penalty}$).

A.3.5 Cache miss time

The miss time t_{miss} is the sum of the hit time (when the cache is accessed again after the miss) and the miss penalty (that accounts for the processor not finding the data and retrieving it from memory).

$$t_{miss} = t_{hit} + t_{penalty}$$

A.3.6 Cache Average Memory Access Time

The Average Memory Access Time $AMAT$ is the weighted average between the hit time and the miss time.

$$AMAT = f_{hit} \cdot t_{hit} + f_{miss} \cdot t_{miss}$$

The miss time can be replaced using $t_{miss} = t_{hit} + t_{penalty}$ to obtain

$$\begin{aligned} AMAT &= f_{hit} \cdot t_{hit} + f_{miss} \cdot t_{miss} \\ &= f_{hit} \cdot t_{hit} + f_{miss} \cdot (t_{hit} + t_{penalty}) \\ &= f_{hit} \cdot t_{hit} + f_{miss} \cdot t_{hit} + f_{miss} \cdot t_{penalty} \\ &= (f_{hit} + f_{miss}) \cdot t_{hit} + f_{miss} \cdot t_{penalty} \end{aligned}$$

By definition, the sum of miss and hit rate is 1, hence we can rewrite the average access time as

$$AMAT = t_{hit} + f_{miss} \cdot t_{penalty}$$

This formula highlights the parameters we can try to optimise to reduce the average memory access time, i.e. hit time, miss penalty and miss rate.

Unified caches

If we consider unified caches, the formula to compute the $AMAT$ has to be modified a bit to consider the fact that the cache is accessed for data and instruction, which have different hit rates. In particular we obtain

$$AMAT = \frac{N_{data}}{N_{total}} \cdot (t_{hit} + f_{miss,data} \cdot t_{penalty}) + \frac{N_{instruction}}{N_{total}} \cdot (t_{hit} + f_{miss,instruction} \cdot t_{penalty})$$

where

- N_{data} is the number of accesses to get data.
- $N_{instruction}$ is the number of accesses to get instructions.
- N_{total} is the total number of accesses to the cache.

Also notice that usually, the miss rate of data accesses is much higher than the one of instruction accesses because instructions are usually stored in contiguous areas of memories, hence the cache can better exploit spatial locality.

L2 and L3 caches

If we consider L2 and L3 caches, the $AMAT$ has to be refined replacing the miss time with the access time to the upper level cache. In particular the miss penalty $t_{miss,L1}$ at level $L1$ in

$$AMAT = t_{hit,L1} + f_{miss,L1} \cdot t_{penalty,L1}$$

can be replaced with the $AMAT$ of the L2 cache to obtain

$$\begin{aligned} AMAT &= t_{hit,L1} + f_{miss,L1} \cdot t_{penalty,L1} \\ &= t_{hit,L1} + f_{miss,L1} \cdot (t_{hit,L2} + f_{miss,L2} \cdot t_{penalty,L2}) \\ &= t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1} \cdot f_{miss,L2} \cdot t_{penalty,L2} \end{aligned}$$

The product between the miss rate of the L1 and L2 caches is called global miss rate and represents the frequency at which a data isn't found neither in the L1 cache nor in the L2 cache. If we call $f_{miss,L1L2}$ the global miss rate we obtain the following formula to compute the average memory access time

$$AMAT = t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1L2} \cdot t_{penalty,L2}$$

The same procedure can be repeated with the L3 cache, hence we can replace $t_{penalty,L2}$ with the $AMAT$ of L3 and obtain

$$\begin{aligned} AMAT &= t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1L2} \cdot t_{penalty,L2} \\ &= t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1L2} \cdot (t_{hit,L3} + f_{miss,L3} \cdot t_{penalty,L3}) \\ &= t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1L2} \cdot t_{hit,L3} + f_{miss,L1L2} \cdot f_{miss,L3} \cdot t_{penalty,L3} \\ &= t_{hit,L1} + f_{miss,L1} \cdot t_{hit,L2} + f_{miss,L1L2} \cdot t_{hit,L3} + f_{miss,L1L2L3} \cdot t_{penalty,L3} \end{aligned}$$

where, as before, $f_{miss,L1L2L3} = f_{miss,L1} \cdot f_{miss,L2} \cdot f_{miss,L3}$.