tricky2.453Processor tasks on Exceptionssection*.69

Computer Architecture
Prof. Paolo Ienne

Arthur Herbette

October 2025

Contents

Introduction

This document is the note I have written during and outside of the course, all the information here is directly taken from the course, slides, etc... However mistake can happen, if you see some mistake or see something that is not clear, feel free to ping an issue on the github LectureNotes, or email/telegram me.
Disclaimer in this course, when we say *high-level language* we mean a language that is compiled/interpreted for instance: `C` is a high-level language here.

Content of the course

The course is divided into three parts:

- **Part I: Processors and ISA**
  What is a processor? How can we design one? How do programs look like when they are executed?
- **Part II: I/Os and Exceptions**
  What is arounf a processor to make a full computer? How the processor exchanges information with the rest of the world?
- **Part III: Memory Hierarchy**
  Processors are fast and memory is slow -how can one combine the two? How can one protext the data users in memory
- **Parti IV: Instruction-Level Parallelism**
  What makes a good processor? How real processors achieve ever increasing performances?
- **Part V: Multiprocessors**
  What are the basic challeng of connecting many processors together? What changes from a single processor system?
- **Parti VI: Rudiments of Hardware Secrurity**
  How can a hacker exploit what we have built in the previous parts to attack a system? How physics helps jeopardizing security?

Literature

The course will have two books for the literature which are the same as the one for fds :

1. Digital Design: Principles and Practices John F. Wakerly
2. Computer organization and design: The hardware software interface David A. Patterson, John L. Hennessy

Part I

Midterm

Chapter 1

Processors and instruction set architecture

1.1 Instruction set architecture

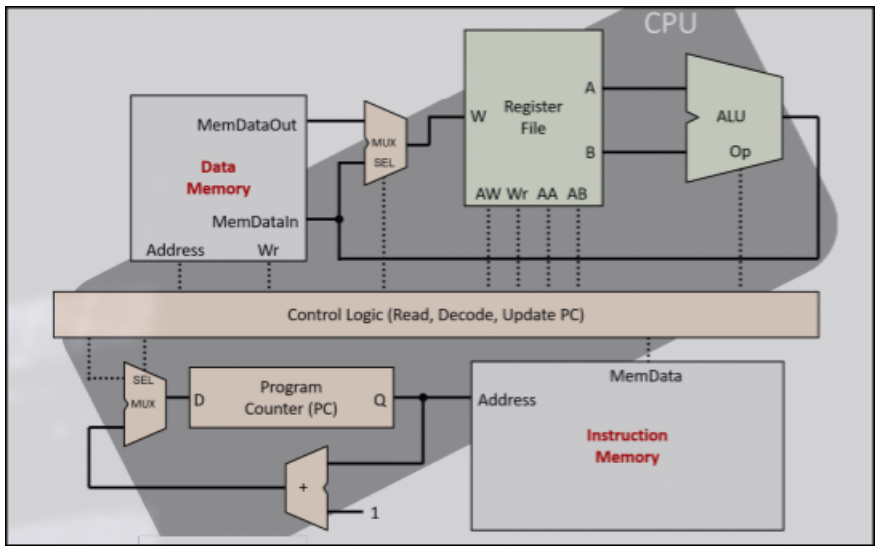
The goal for the beggining of this course is to go from "high-level" perspective to the bottom of the iceberg. First let us look at a piece of code (`c`):

```c
int data = 0x00123456;
int result = 0;
int mask = 1;
int count = 0;
int temp = 0;
int limit = 32;
do{
   temp = data & mask;
   result = result + temp;
   data = data >> 1;
   count = count + 1;
} while (count != limite);
```

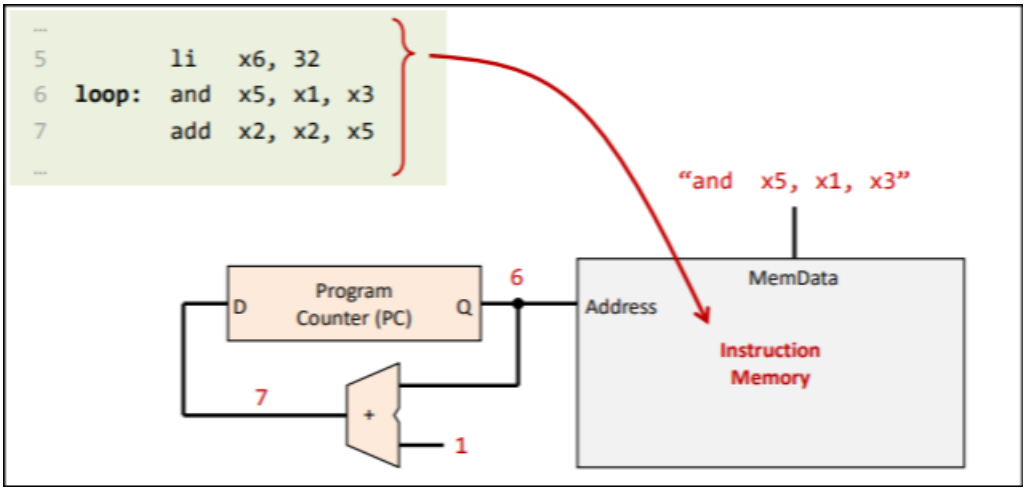
Here we can see that we have variable with expressive names (that we can choose). each variable has a type, the computation we are doing `result + temp` looks like a mathematic formula, the control flow we are using is very intuitive.
In the case of those high-level language, we have an "unlimited" number of variables which supports any type.
If we wanted to convert this code into Assembly code we would have this:

```
    li x1, 0x00123456
        li x2, 0
        li x3, 1
        li x4, 0
        li x5, 0
        li x6, 32
loop:
        and x5, x1, x3
        add x2, x2, x5
        srli x1, x1, 1
        addi x4, x4, 1
        bne x4, x6, loop
```

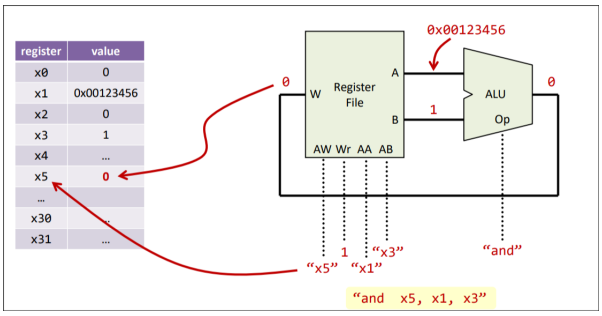
As we can see we have a much more rigid format: we really have a sequence of numbered instructions that is executed line by line. For each instructions, we have an *opcode* that defines the effect of the instruction. Each *variable* has a fixed name and we only have one form of control flow. The question to ask is why did we do that?
The answer of thie question lays is in the architecture of the processor:



how it works: the processor fetch the instruction at the address of the program counter (PC) and launch it to the control logic

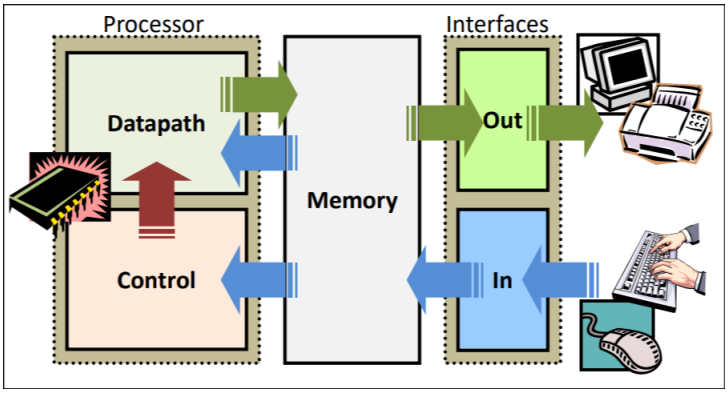


After that the instruction has been fetch, it is processed in the Control logic and then read/write etc... into the register file, and give the information (the opcode) to the ALU for it to know which operation to perform.

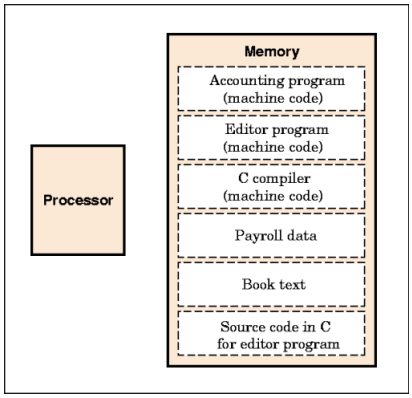


The five classic components of a computer

For an every day computer you need four other components other than the control components, you need to have a memory to store data (bigger than 32 word registers), you need to take input from the outside worlds (Internet, bluetooth, a keyboard, mouse ...) and also output something to the outside word. On top of that, you need all of that to communicate $\implies$ you need a data path.



Okay, we have memory, we have input output and a place to compute everything, but what do we need to compute? Where is the program that is being executed? At the moment we have a place for the data but not for our program so how do we do it?
We store the program in the same memory than the one for the data. This is called a *Unified Architecture* (On the other hand, an architecture that have two seperates memory, one for the instruction and one for the data is called a *Harvard Architecture*). This is a **Key concept to computer science**, our instruction (therefore program) are represented as numbers (juste like data).



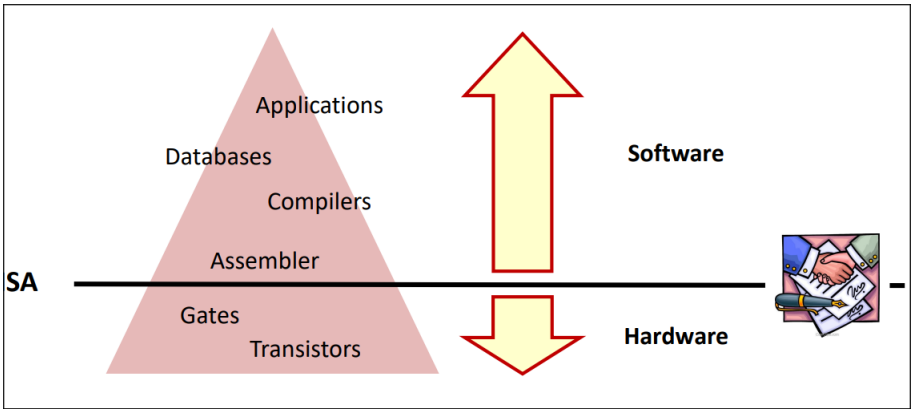
Now a good question to have is: how to decode and encode those instructions and in the mean time, also what makes a good encoding?

A good encoding would be one that allows us two minimize the ressource in hardware and also is the fastest. This is where `RISC-V` comes in the play!! `RISC-V` is an instruction set architecture as like many others for instance x86, x64 for the most famous and used one.
The difference between assembly language and high-level language is in the "*translation*", for the assembly language, we use an **assembler**, for a high-level language (a compiled one), we use a **compiler**:

- Assembler can easily translate from code to binary code (this is what the instruction set tells us to do). All we need to do is the look up in the table and translate
- A compiler on the other hand cannot look up in a table, it has to translate the code into Assembly code to be translated, but compiled the code into Assembly is a very hard thing to do, you have to find the best way or at least, try to find the best way to say the same thing but in assembly.

1.2 Instruction set architecture: Branches, Function and stack

The main goal that ISA does is to put a **Contract** between the hardware and the software: If you are an hardware person, all you care about is the make your processor the fastest on the ISA. If you are a software person, you don't need to worry about the hardware behind anything, you only care about the software that you are building. This ISA gives a level of abstraction which makes it easier to develop better software/hardware.



As we have seen in cs-173, arithmetic and logic operation are quite easy to understand and use in RISC-V. But here are some facts to know about them:

- Immediate constant takes at maximum 12 bits. The reason behind this is that the immediate part of the instruction is directly stored in the instruction, this means that there are 12 bits of the instruction that are reserved for the immediate part. Imagine having for instance a 30 bits immediate, then you would only have 2 bits for: the opcode, result register, input register ...
- A way to go around this is to use the `.equ, num` and then to use `lui` directly on `num`. (this is possible because the assembler will directly translate the one line instruction into a three lines instruction).
- Register `x0`, this register is **always** zero (by definition), you can write anything to this resgister, the value in it will always be zero. This can be useful in a lot of case, it happens quite often that we need a zero in a instruction and the only way to do so would have been to `li` a register to 0 and then calling the instruction. Therefore, the `x0` register allows us to save instructions

An if-then-else

To be able to do an if and else cause will need some branches, for instance if we wanted to translate the code:

```
if (x5 == 72) {
   x6 = x6 + 1
} else {
   x6 = x6 - 1
}
...
```

Into RISC-V: it would look like this:

```
.text
        li x7, 72
        beq x5, x7, then_clause
else_clause:
        addi x6, x6, -1
        j end_if
then_clause:
        addi x6, x6, 1
end_if:
        ...
```

As you can see jump and branch are really similar however, there is a universal distinction between them:

- Jumps → **unconditional** control transfer instructions
- Branch → **conditional** control transfer instructions

However this is not the case for every assembly languages, for instance in x86, everything is defined as a jump.

A Do-while loop

A do while loop in c

```
do {
   x5 = x5 >> 1
   x6 = x6 + 1
} while (x5 != 0);
...
```

A do while loop in risc-v

```
.text
loop:
        srli x5, x5, 1
        addi x6, x6, 1
        bnez x5, loop
...
```

1.2.1 Functions

In our high-level code, we usually use function to organized our code (Scala...) (those function can also be called methods, procedure dependeing of the context).
What we would like is also to have function in assembly so that we don't have to write the same code always. What a function would look like is:

1. Place arguments where the called function can access them
2. jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function
5. Communicate the result value back to the calling program
6. realease any local storage resources
7. Return control to the calling program

That sound pretty hard to do so let's do it step by step. First, the second and seven steps (I know). What we need is to jump to the function and the return. This is fairly easy to do, all we need is to call the jump instruction. For instance, let's call the function two times. This would looks like this

```
sqrt:
        ...
        j back
```

And the main would look like this:

```
main:
        ...
        j sqrt
back:
        ...
        j sqrt
back2:
        ...
```

However, isn't there an issue? what would happen if we tried to run this code?
The answer is that this would lead to an infinite loop. the `sqrt` function doesn't know about the fact that there are more than one back. The solution to this problem is to:
when you called the function, you store the current PC +4 (to go to the next line) to a register (for instance `x1`). You then, call the function, do the computation there **and then** you rejump to the address store in the register `x1`.

Jump and link

There is instruction that allows us to do this, those instruction are called jamp and link `jal`, and the other one is called jump to the address specified in a register `jr`, however we said before that we only use `x1` for the return address so why don't we make an instruction that directly jump to this address: `ret` (which stands for return I think).
However what we have to be careful with here is that the `x1` register is not preserved accorss the call (this is not something that is known for now but let me explain it shortly). What we will want to do is the call function inside function (have call inside call inside call etc ...) however every time we make a call to a function, the x1 register will be overwritten: every time you jump and link, you store in the return address register the pc +4. However this is not currently a problem, we will solve it later.

Acquire storage ressource the function needs There is a lot of way to do this. The first way to do so is to juste allocate like 10 registers to the current function and the rest to the function that is called. for instance if we have this code:

```
main:
        ...
        jal sqrt
        ...

        ...
        jal sqrt
        ...
ret

sqrt:
        ...

        add x5, x7, x8
        jal round
        sub x6, x6, x5
        ...
        ret

round:
        ...
        addi x10, x11, 3
        ...

        ret
```

You see that the round procedure only use the register `x10` to `x15`. and that sqrt the one from 2 to 9. We can clearly see that this is not scalable, so we need another solution.

1.2.2 The stack

The **stack** is the solution!! Fisrt what is the stack:

Definition 1.

- The stack is a empty region in the memory
- We use the register `x2` (also called `sp`) to store the address of the end of the used region
- If we are using all variables and we still want to make a call to a function, we need to store in the stack our variable before calling the function and then restore our variable from the stack.

The complexity of this is to understand the order of what is needed to be stored or not. For instance if you have a function that is being called from above. We have to be sure that we don't overwrite the values from the function that is above. to do so, we store the value in the stack and restore them afterward. (only the register that we are changing). to do so we have to dynamically allocate more space in the stack.
Here is an example:

```
        ...
        addi sp, sp, -8
```

```
        sw x8, 0(sp)
        sw x9, 4(sp)
        ...
        #we have here free use of x8 and x9
        ...
        lw x9, 4(sp)
        lw x8, 0(sp)
        addi sp, sp, 8
```

However, do we need to store all the register? how do we return something, how do we pass arguements to a function. To do so we agree to use some register as arguement, return register, other for return address, stack pointer, temporaries, saved... I strongly advise to go read the RV32i Reference Card.
So what do we still need? we are currently able to jump to function, return from the function, acquire storage resources, perform the desired stack of the function, All we need is the arguement and return values. To do so is very simple, as I said before we can:

- Use some particular registers, both for the **arguments** and for the return **result**.
- We can do it ad-hoc ...
  - `sqrt` gets the arguement in `x5` and returns the result in `x6`
- Or we can have some convention
  - All function pass arguements in register `x10` to `x17` and return the result in `x10`
- Can this be insufficient? **More arguments** than allocated registers? What if we have 10 arguements

Option 2 If we don't have enough registers, we can just put them in the task right? we know that the stack is unlimited (in theory), all we would need is to do more work (allocate space, storing, loading etc ...)
To do so we can use another register: `fp` or `x8` in risc-v which point to the same location as sp on entry.
This make the code more readable because:

- `sp` changes inside the function and so do relative offsets
- offests with respect to the `fp` are **fixed**

The use of the fp register is **optional** and even varies among users and compilers. (I personnaly didn't use it during lab 1, I only used the registers that are reserved).

Register	Mnemonic	Description	Preserved across Call?
x0	zero	Hard-wired zero	—
x1	ra	Return Address	No
x2	sp	Stack Pointer	Yes
x3	gp	Global Pointer	—
x4	tp	Thread Pointer	—
x5	t0	Temporary/alternate link register	No
x6–x7	t1–t2	Temporaries	No
x8	s0/fp	Saved register/Frame Pointer	Yes
x9	s1	Saved register	Yes
x10–x11	a0–a1	Function Arguments/return values	No
x12–x17	a2–a7	Function Arguments	No
x18–x27	s2–s11	Saved registers	Yes
x28–x31	t3–t6	Temporaries	No
pc		Program counter	—

1.3 Memory and Addressing Modes

1.3.1 Memory

Memory is an incredibly important component of a computing system:

- We store our **programs** in it
- We store our **data** in it
- It is often through memory that we will **receive data and send out data**

Memory is a reccurent ropic in this course, we have already seen it with the stack however the type of the memory is also an important topic:

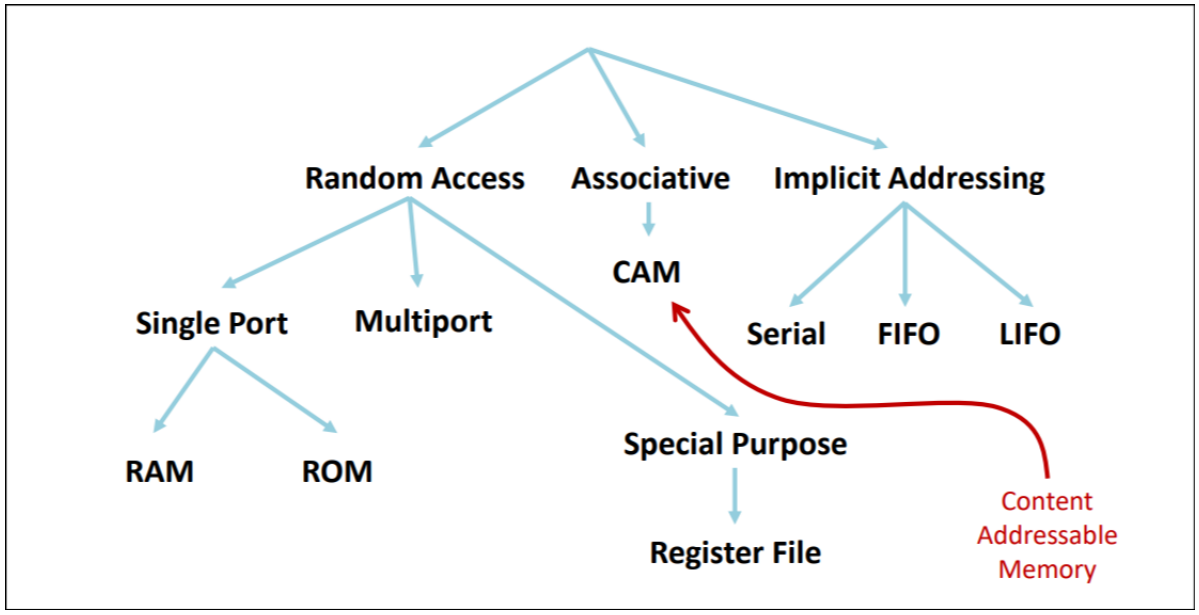
- Memory can be **very slow** $\implies$ Caches
- Memory is *finite* (relatively small) $\implies$ Virtual memory
- Memory can make an **ISA too complex** $\implies$ pipelining

Types of memory There is a lot of different technologies for memory:

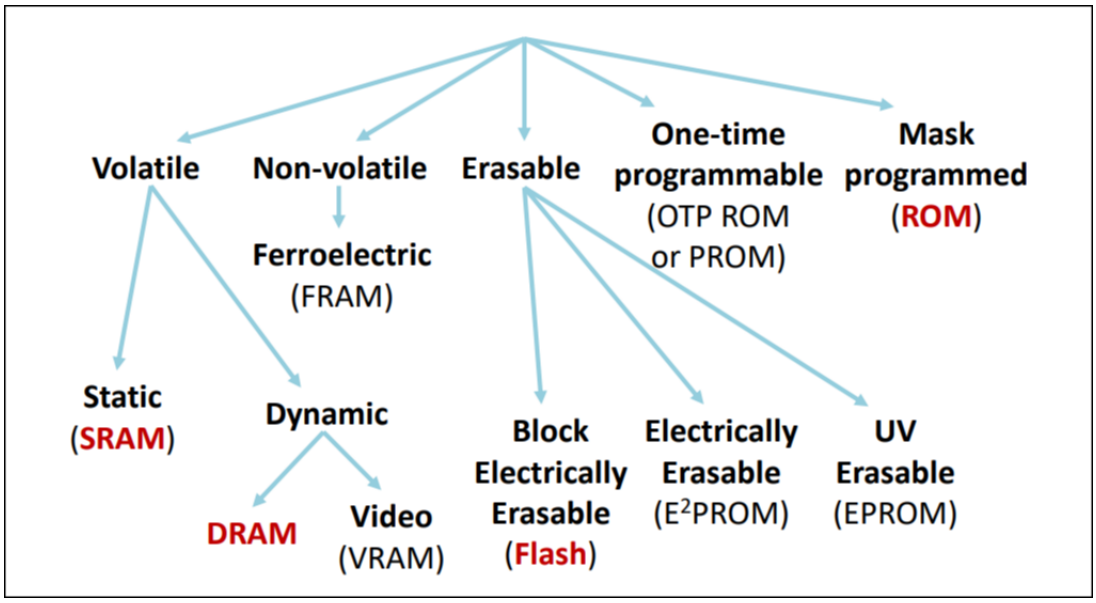
- SRAM, DRAM, EPROM, Flash, etc.

Each of those has a variations in **capabilities** therefore also in how we use them, memory change by:

- Capacity, density
- Speed
- Writable, permanent, reprogrammable

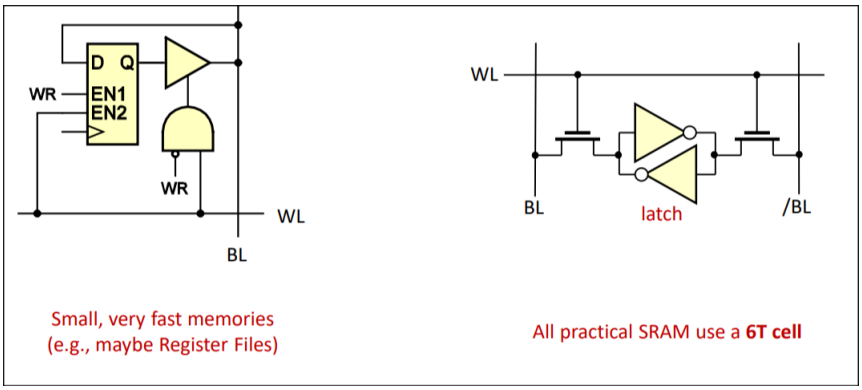


We have here all the type of memory that can be used, What we use when we program are Random access memory, this means that the memory can be accessed with an address. In the tree of random access memory we have:



The basic structure behind those memory are DFF, (D flip flop). which are stacked one on another in a n times 4 grid. Each flip flop looks like this:

SRAM SRAM stands for **static random access memory**. It sores data with flips flops which makes it faster than DRAM (which we will see later) but more expensive. We use it for CPU caches and for the register file



So here we have to make a difference between the boolean system and the electrical components. The circuit on the left is a desastre in term of electrical components it has approximatevely 20 transistors which makes it **slower and costlier**. The real way to do SRAM is with the right circuit. However this looks bad, normally it is forbidden to have a closed loop in a circuit! We are forbidden to have a loop in our circuit without a flip flop in it, you cannot have a loop inside a combinational circuit. So this is a big special thing for us however, this "works", it is compatible there is no issue in the circuit. The issue we have is that:
Imagine putting a one on the left or right part of the circuit $\implies$ the value cannot be changed, it is stucked there for ever. This looks really good because one **NOT** gate costs us only two transistors so the memory (loop) costs us only 4 transistors. But we still need to write and read from the memory, to do so we had the two transistors (see on the image) which also us to let the memory live on its own (when the transistors are open) **or** to be connected to the world.
If I want to see what is on the memory I put one in the word line (WL) and I will get the value on the bit line.
The question now is how to write? As said earlier, now we have a signal that is stored in there but

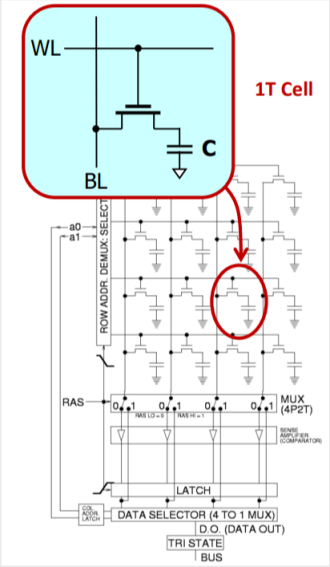
it is stored for infinity.
The only way to write is to "*shout louder than the current signal*". Imagine we currently have a 1 as the output of the latch and I want to put a 0. If I shout 0 louder than the 1 while connected, it will have a short circuit... and this is bad. **However** what is going on in fact is the upper not gate will have two inputs, a **loud** 0 and a quiet 1, the not fate will then take the loudest one thus 0.
And now it will take a really short time to the latch to adapt itself to the new value, the short circuit here take the times two the 0 to go trhough two not gates. and then it agrees.

On the other hand we have DRAM:

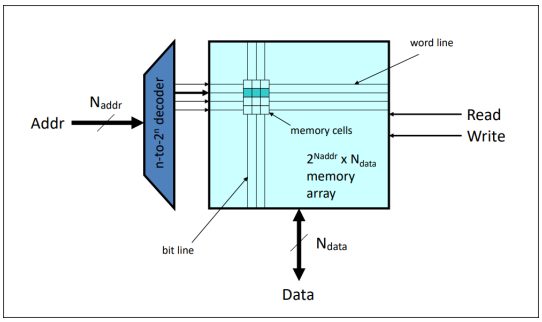
DRAM

- Dynamic RAMs are the densest (and thus cheapest) form of random access semiconductor memory
- DRAMs store **information as charge in small capacitors** part of the memory cell
- First parented in 1968 by Robert Dennard, scaled amazingly over decades and was somehow an important ingredient of the progress of computing systems.
- charges **leaks off** the capacitors due to parasitic resistance $\implies$ every DRAM cell needs a **periodic refresh** (e.g. every 60ms) lest it forgets information.

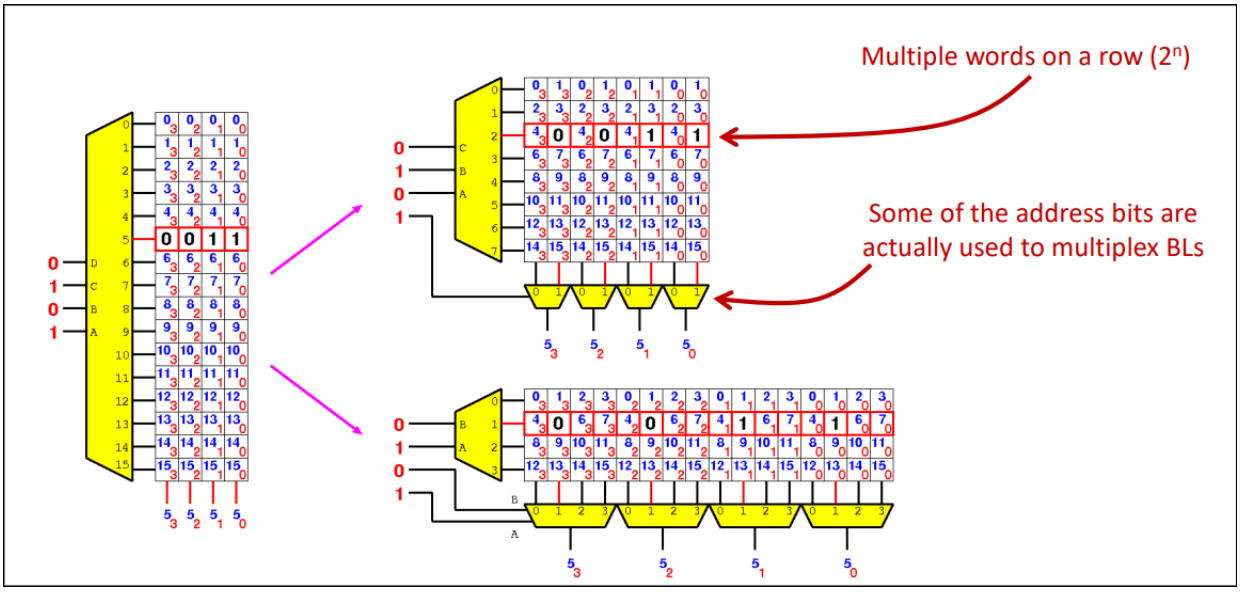
So imagine, if we don't go in each cell every 60ms then we lose the information, but we have other things to do? So how do we do it? - we have someone else refresh them for us. The memory controller is reponsible for refreshing the contents of the DRAM instead of the CPU.



The goal after this is to access those memory celle based on the address we input. The *ideal* way to do so, would be to have one **big** decoder that treats the adress and directly output the information in the memory celle like this:



However life is not always that easy, and there is a lot of way to get the memory celle based on the address, here are some example:



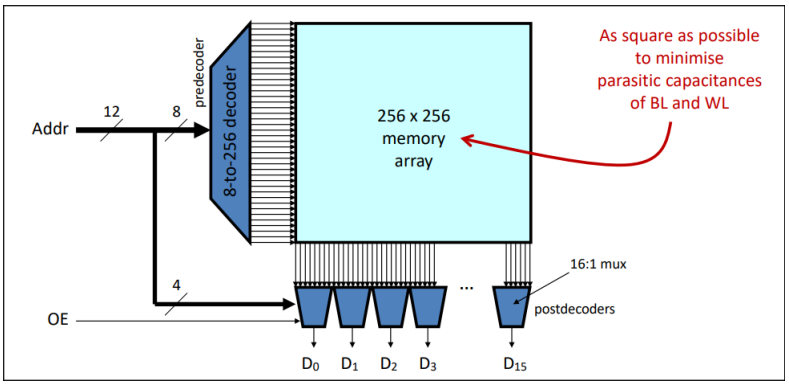
Here we have 3 ways to do so:

- On the left, We have the same way as the *ideal* decoder with one byte per row
- However we can also split up into a grid with more than one **big** multiplexer. This implies that there will be multiple word by row and that the bytes are not necesserly ordered.

The best physical way to create a Random access memory is in a square to minimize parasitic capacitance of BL (bit line) and WL (word line). We want to having it into the most squared possible form because:

- When a word line is activated (row), the bit line carries the data (bit) stored in the selected memory cell to the output circuitry (like a multiplexer or sense amplifier).
- Activating a word line selects all the bits (across bit lines) in that row — this is your selected "word."

Therefore by having the smallest length, we get shorter lines $\implies$ lower parasitic capacitance $\implies$ faster access, lower power, and more reliable operation.



Remark 1. Every time we are looking for a memory cell, we need to charge all row and then all column, the goal here is to minimize the number $x = r + c$ by a fixed area A (where A is the number of cell):
We have that

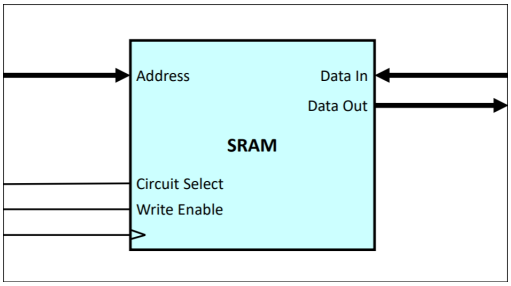
$$A = rc$$
$$\frac{A}{c} = r$$

Which implies that $x = \frac{A}{c} + c$, we are minimizing $(x' = 0)$ this:

$$x' = -\frac{A}{c^2} + 1$$
$$\frac{A}{c^2} = 1$$
$$c^2 = A \implies c = \sqrt{A}$$

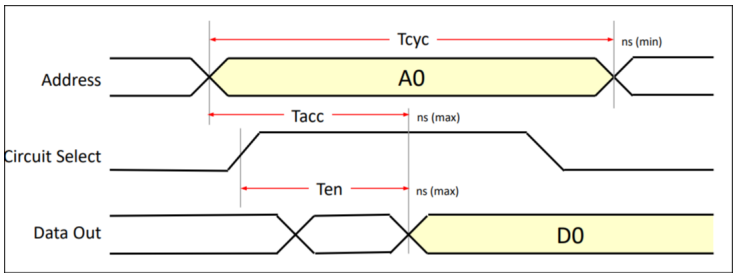
And because we know that $A = rc \implies r = c = \sqrt{A}$ which is a square.

Static RAM typical interface This is the typical synchronous SRAM that we have already seen before:



However we don't always has to be synchronous, we can also be asynchronous for a Read cycle which works like this: **Asynchronous read cycle**

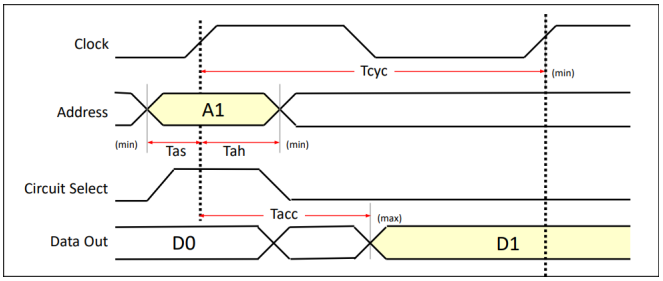
- Enable the memory → assert the address → wait for the data
  - Data out is available after a combinational delay T_{acc} = Access Time
- Maximum frequency is limited by the minimum T_{cyc} (time for a cycle, time for us to be able to change the address)



synchronous SRAM Read cycle

Here this is the other way around, we always wait a rising edge of the clock to do anything. Everything here is working like a flip flop:

- Everything is relative to the clock signal
- Latency is the number of cycles between the address asserted and data available
  - Often one as in this diagram but in some cases (large memories) more

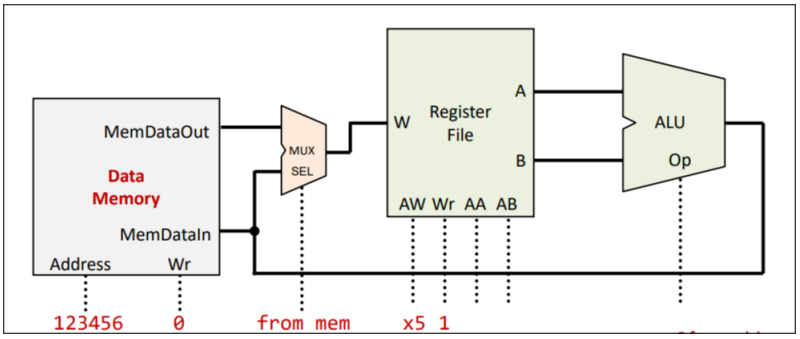


Load and store instrucitons

Now that we have seen how it works, we want to see how to implement a load from the memory into the register file. For instance the following intrsuction:

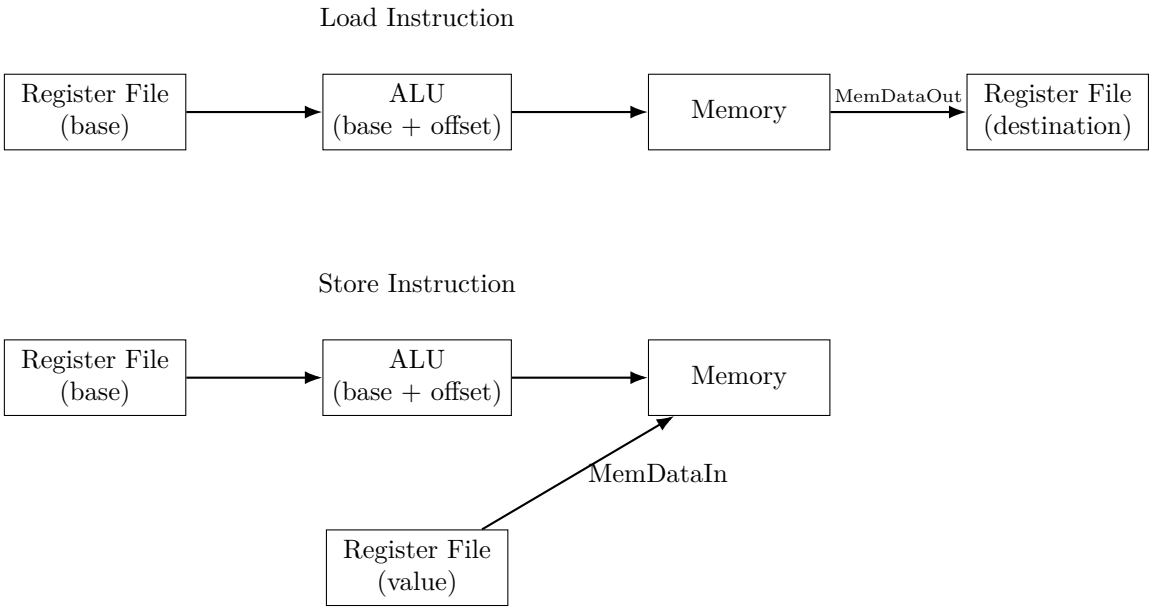
```
lw x5, (123456)
```

This is not a RISC-V instruction but bear with us. (I am not sure that's a saying...)



What we have changed here is the left part now instead of having a loop like ALU → Register file → ALU we break it at the first "→" and add a multiplexer there to be able to interact with the memory.

For the store instruction, instead of using the `MemDataOut` path, we use the `MemDataIn`.
The main diffrence is this:



Remark 2. I had some trouble understand how does the value just pop, but if I understand it right, the register value is stored from the register file into `B` here. We use the `A` port as a address base. This is how it works:

1. IF, Fetch instruction (`sw x2, 8(x1)`)
2. ID, Read `x1` and `x2` from register file
3. EX, ALU compute `x1 + 8` (address)
4. MEM, Store `x2` to memory at computed address
5. WB, Nothing (no register to write for store)

Why RISC-V instructions are so simple?

Remark 3. Here are some example of some instruction that would look correct in RISC-V but is not (for addition):

- Based or Indexed

```
add x0, x1, i5(x2)              #x0 = x1 + mem[x2 + i5]
```

- Auto-increment or -decrement

```
add x0, x1, (x2+)               #x0 = x1 + mem[x2]
```

- PC-relative

```
add x0, x1, 123(pc)             #x0 = x1 + mem[pc + 123]
```

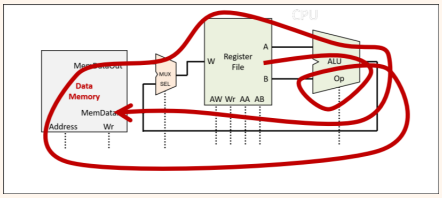
However those instruciton **does not exist** in RISC-V.
RISC-V is designed to have two world: one for accessing memory, and one for the logic/arithmetic etc. We cannot mix them together.
However in x86/x64 we can do this:

```
ADD DWORD PTR [EBX + ESI*4 + 16], EAX
```

This means:

- The `DWORD` means double word which means that we are working on 64 bits number.
- The `ADD` that has only two operand, the reason why, is that the goal of x86 in 1979 was to be the most compact possible, at that time the memory was limited and to be able to write program you had to be careful of the size of the program that you are writing. So they added the constraints that the output of the instruction is stored in the first operand.
  this means we take something in the memory at `[EBX + ESI*4 + 16]`, add it with `EAX` and then store it in `[EBX + ESI*4 + 16]`.

this feels pretty slow and pretty confusing, just try to map this into the CPU we created before, this would look like this:

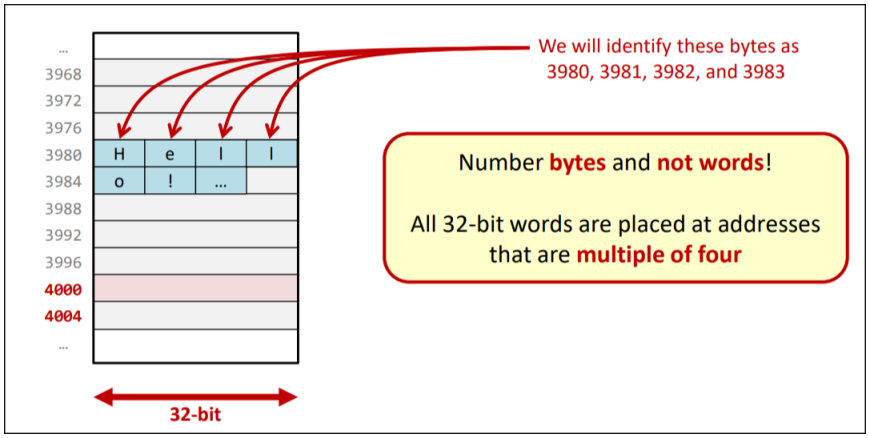


We would need to go like four times to the ALU, the fact is that **this is possible** however it makes it hard to optimize the processor.

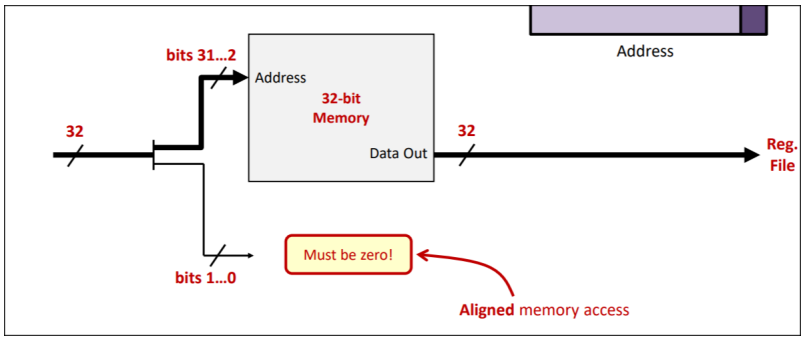
The question behind all this is how does intel can still be as famous as they are now with instruction that looks like this and that cannot really be optimized? There is still the majority of the processor to this day that are intel processor (even if amd is better...), we will see this in a couples of weeks.

Byte addressed memory

Almost all ISA today are use byte addressed memory. byte are quite important, disks are organized in bytes, network packets are bytes, **ascii** are byte. A lot of data are represented as byte so using an word addressed memory wouldn't be very efficient here, we would either loose a lot of time looking for the right byte, or loosing a lot of space by putting byte in word address (losing the three other bytes).
The solution is to no change the way memory is placed **but changing the label**.

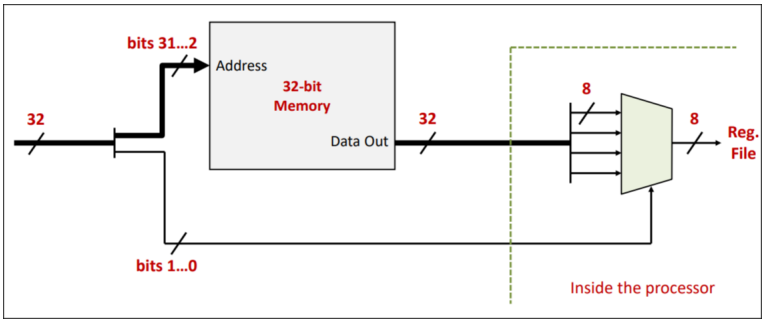


Loading a word (`lw`) and Instruction For instance, if we are interested in word, we cannot look for the word at address 3981, this would'nt be a word.



As said before when loading a word, it has to be a multiple of 4 so we only care about the 30 most significant bits. The two least significant bits is checked wether there are zero or not and if there are not zero, we would like to **throw an exception** which we will see how in a couple of weeks.

Loading bytes (`lb`) Here what we are doing is the same as what we did before, we are looking for a word (which is the 30 first bits), and then in the word we choose which 8 bits we wants, this would look like this:



Remember when we changed the shape of the memory, how we choose which bit to take; we are doing the **same** here too. The difference is that this part is only in the processor, circuit wise this change nothing.
What is good here is that by just adding a multiplexer we can add a lot of instruction in the ISA.

1.4 Arrays and data structures

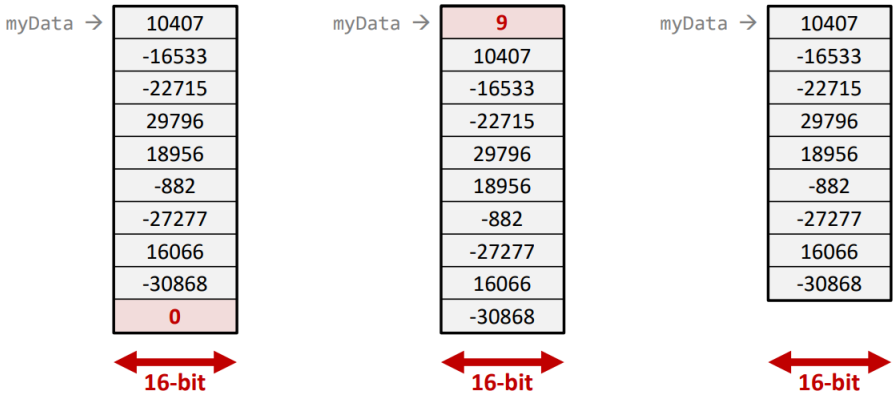
Data structures is one of the main concept in computer science, even in this course we have already talk about it (the stack).

Arrays in high-level languages

```
val myData: Array[Short] = Array(10407, -16533, -22715, 123133, 12512)
```

Here we have a sequence of number that are indexed, the question however is how do we store them?
We have a lot of ways to do so (three), we can:
have a pointer to the first element, and the having the other element following this one. With this method the issue is when does the array stops? here's three example of how to store arrays:

1. One way to do so is to put a *null* element at the end of the array so that we know that this is the end of the array. The definition of string are in fact an array of char with the 0 char at the end.
2. A whole other way is to have, at the start of the array the length of the array. You have a pointer at the start of the array which is the length of the array and then you do your computing as usual
3. Another way with this is just to to do nothing. Having a pointer to the start of the array and then hoping that the programmer knows what he is his doing. `C` Arrays for instance are built like this.



Adding Positive elements

To add all the positive elements in an array of signed 16-bit integers we would:

- At call time → `a0` points to the array (and, in type 3, `a1` is the length)
- At return time → `a0` contain the result

The result for the type 3 (written in `c`):

```
short add_positive(short myData[], int N) {
        short  sum = 0;
        for (int i = 0; i < N; i++) {
                if (myData[i] > 0) {
                        sum += myData[i];
                }
        }
        return sum;
}
```

Adding positive elements (Type 1)

For the first type let us write the code in RISC-V:

```
add_positive:
        li t0, 0  #t0 will hold the sum (initialized to 0)

next_short:
        lh t1, 0(a0) # Load short (half-word) at address a0 into t1
        beqz t1, end  # If t1 is 0 (null short) we are done
        bltz t1, negative  # if t1 is negative ignore
        add t0, t0, t1  #Add t1 to the sum (t0)
```

```
negative:
        addi a0, a0, 2  # move array pointer (a0) by sizeof(short) to
            the next element
        j next_short  # repeat the loop
end:
        mv a0, t0  # move the sum (t0) into a0 as the return value
        ret # Return the caller
```

Adding positive element Type 2

```
addi_positive:
        li 01, 0 #t0 will hold the sum (initialized to 0)
        lh t1, 0(a0) # t1 will count the elemetnts to process
        add a0, a0, 2  # Move array pointer (a0) to the first real
            element

next_short:
        beqz t1, end  # If t1 is 0 (no more elements), we are done
        lh t2, 0(a0) # Load short (half-word) at address a0 into t2
        bltz t2, negative # If t2 is negative, ignore
        add t0, t0 t2  # Add t2 to the sum (t0)

negative:
        addi a0, a0, 2 # Move array pointer (a0) by sizeof(short)
        addi t1, t1, -1 # Decrement the counter of elements to process
        j next_short  # repeat the loop
end:
        mv a0, t0 # Move the sum (t0) into a0 as the return value
        ret # Return to caller
```

Adding positive element Type 3

```
addi_positive:
        li 01, 0 #t0 will hold the sum (initialized to 0)
        mv t1, a1 # t1 will count the elemnts to process (a1)

next_short:
        beqz t1, end  # If t1 is 0 (no more elements), we are done
        lh t2, 0(a0) # Load short (half-word) at address a0 into t2
        bltz t2, negative # If t2 is negative, ignore
        add t0, t0 t2  # Add t2 to the sum (t0)

negative:
        addi a0, a0, 2 # Move array pointer (a0) by sizeof(short)
        addi t1, t1, -1 # Decrement the counter of elements to process
        j next_short  # repeat the loop
end:
        mv a0, t0 # Move the sum (t0) into a0 as the return value
        ret # Return to caller
```

Adding positive elements (variation on type 3)

Let us add positive elements in an array of signed 16-bits integers:

- At call time → `a0` points to the arrays and `a1` is the length of the array
- At return time → `a0` contains the result

The way od doing it is to incremented the index of the array:

```
int i = 0;
while (i < N) {
        if (myData[i] > 0) {
                ...
        }
        i++
}
```

This is equivalent to:

```
int i;
for (i = 0; i < n; i++) {
        if (myData[i] > 0) {
                ...
        }
}
```

Here we see that we have a variable `i` which is incremented by one, therefore, the way of doing this if we were to be compiled would be by having a variable that is incremented by 1 in every loop **then** be multiplied by the size of the data (2 bytes).

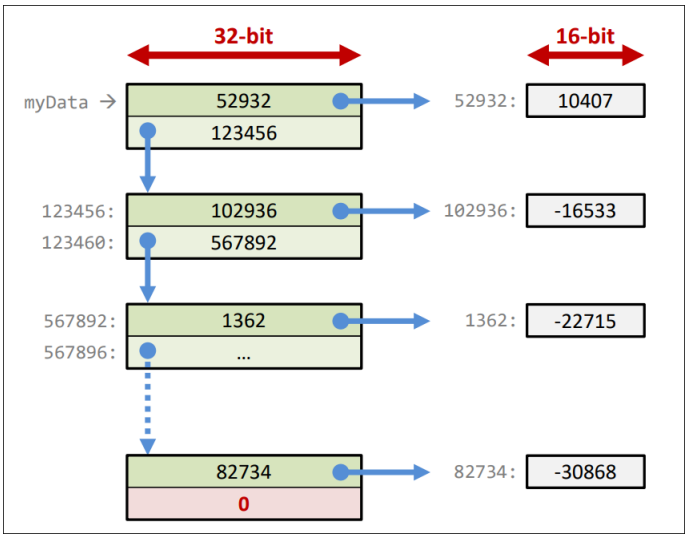
```
addi_positive:
        li t0, 0 #t0 will hold the sum (initialized to 0)
        mv t1, 0 # t1 will hold the array index

next_index:
        beqz t1, end  # If index >= number of elements , we are done
        slli t2, t1, 1  # t2 = offset of the element as index (t1) *
            sizeof(short)
        add t2, a0, t2 # Address of the element = myData (a0) + offset
            (t2)
        lh t3, 0(t2) # load short (half-word) at address a0 into t3
        bltz t3, negative #if t3 is negative, ignore
        add t0, t0, t3 # Add t3, to the sum (t0)

negativ:
        addi t1, t1, 1 #Increment the counter of element to process
        j next_index  # repeat the loop
end:
        mv a0, t0 # Move the sum (t0) into a0 as the return value
        ret # Return to caller
```

Which one is better?

We have now two different way to do the same type (type 3), however which one is faster? this question can be easily answers:
the first one that we have written has less intrustion $\implies$ faster. (this is not always that simple)
But this points out an issue, we need **good compiler**, we need a compiler that can translated our code into the **fastest assembly code possible**.

Linked list

Another way to store data is with a linked list:



As we can see here this is the same principle as what we did before, we store for each element the current address of the value **and** the address of the next value. To iterate through this list all we have to do is to go to the current value get the value via the address and then take the next iteration with the second address.
What is good about this is that:

- Insert element in the array is very easy

However there is a lot of bad thing here:

- For each value we have to use 64 bits of addresses which is a lot
- iterate through the list seems nice however are all the instruction truly equal?

For instance imagine we wanted to recreate the same code as the one we wrote for the other arrays:

```
add_positive:
        li t0, 0 # t0 will hold the sum
next_element:
        beqz a0, end # If address of next element (a0) is zero, we are
             done
        lw t1, 0(a0) # Load address of actual data into t1
        lh t1, 0(t1) # Load short (half-word) at address t1 into t1
        bltz t1, negative # if t1 is negative, ignore
        add t0, t0, t1  #  Add t1 to the sum (t0)

negative:
        lw a0, 4(a0) # Load address of next element into a0
        j next_index
end:
        mv a0, t0 # Move the sum (t0) into a0 as the return value
        ret # Return to caller
```

As we can see here: this is not much more complex. but is it more efficient?
no. The instruction of loading and storing are way **slower** than the other instruction, the fact that this way of computing leads to a lot more of load makes it way slower.loading byte

1.5 1.e: Instruction Set architecture Arithmetic

Notation

- Number (represented on a specific no. of digits/bits)

$$A = A^{(n)} = A^{(m)}$$

- Number (in binary or decimal)

$$A = A_{10} = A_2 = A_{2c}$$

- Individual digits (bits)

$$a_{n-1}, a_{n-2}, \dots, a_2, a_1, a_0$$

- Digit string (representation)

$$< a_{n-1}a_{n-2}\cdots a_2a_1a_0 >$$

Numbers

we usually care for three types of numbers:

- **Integers** (signed and unsigned)
- Fixed point

$$0.12, 3.14, 10131412125 12.5124213$$

  - Essentially integers with **implicit 10^k or 2^k scaling**
  - Extremly important in practice (most signal processing is fixed point)
- **Floating point**

$$3.14E3, -2.4E-1$$

Remark 4. As we have seen in it fds, we feel like fixed point are just some useless number representation but this is **false**. As said before, in signal processing we use a lot of number that needs to be *pointed* (not integers), but it also need to be **fast** as for us to be able to watch a live twitch or a youtube video... We need to do a lot of computation the fastest way possible. Floating point are pretty bad at this, addition using floating is much more slower than integers addition, we know that fixed point are just integers disguised as rational number. That's the reason why we use fixed point representation.

Unsigned Integers

$$A = \sum_{i=0}^{n-1} a_i R^i$$

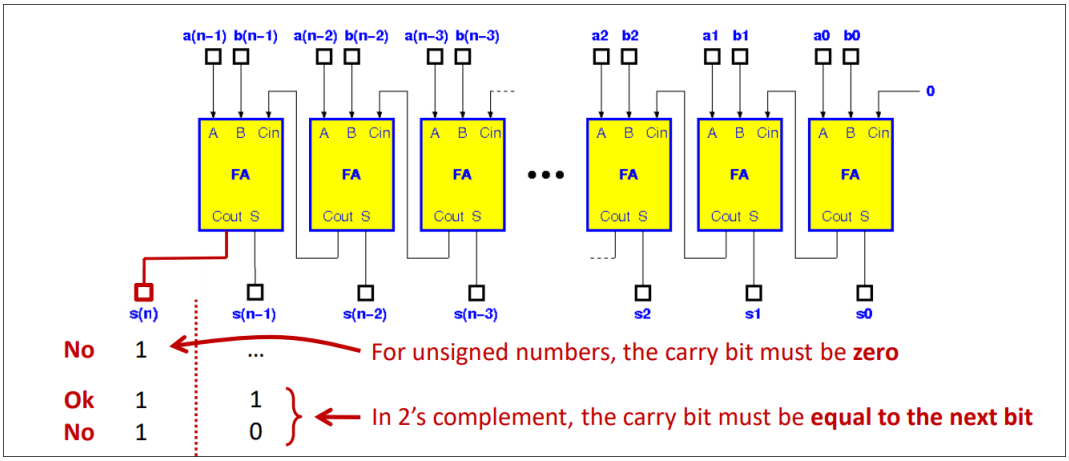
Remark 5. For the next part of this course I am gonna skips this because it is a big review of what we have already seen in fds, I am just going to write what I find intresting **for me** which is not necessarly the most important thing.

Addition is unchanged from unsigned

As we can see (remember of the table) addition for signed number (in two's complement) and unsigned number is the same, this allows us to have **only** two instructions (`add` and `sub`) without any `addu` like instruction.
This is one of the reason why we use 2's complement as the universal representation of signed integers today.

Overflow in hardware

In hardware, **carry out** is the only missing bit from the **complete** result
We can think of overflows as a **tuncation** problem:



Overflow in software

Some architecture (e.g., **x86**) gives us the **carry bit** in a special register (a **flag**)

$$\rightarrow \text{overflow detection is the same as in hardware}$$

Other modern architecture gives us **only the result** of the addition (e.g., **RISC-V**).
The detection is usually based on the following observations:

- If addition of **opposite sign number** $\implies$ magnitude can only reduce $\rightarrow$**no overflow possible**
- If addition of **same sign number** $\implies$ overflow is possible but the sign of the result will appear wrong.

$A + \overline{A} = -1$

As we have seen here

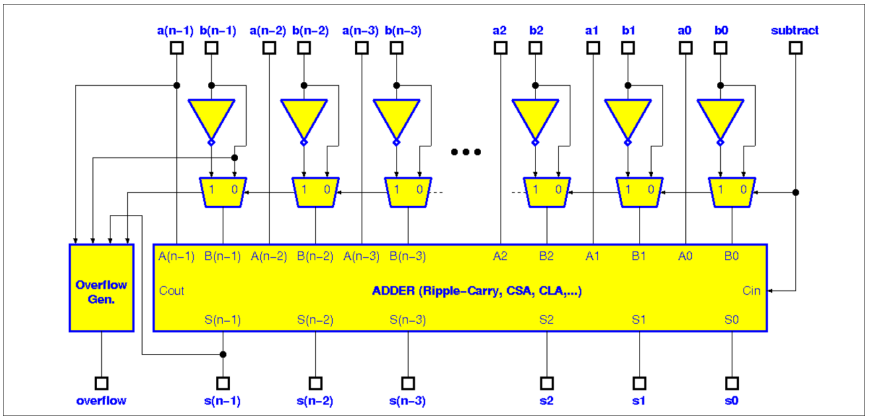
$$-A = \overline{A} + 1$$

To prove it:

$$\left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right) + \left(-\overline{a_{n-1}}2^{n-1} + \sum_{i=0}^{n-2} \overline{a_i} 2^i\right)$$
$$= -(a_{n-1} + \overline{a_{n-1}}) \cdot 2^{n-i} + \sum_{i=0}^{n-2} (a_i + \overline{a_i}) \cdot 2^i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i$$
$$= -1$$

Two's complement Add/Subtract Units

With this proprety it becomes very easy to compute subtraction as it is just the use of the addition part but with the inverse + 1. This can be done by having a c_{in} at the beggining of the adder and to have a multiplexer to choose between the b_i or $\neg b_i$



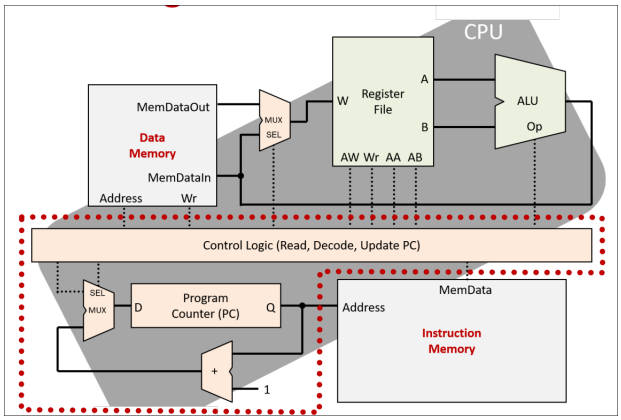
as we can see this allows us to put the substraction and addition into the same module.

Chapter 2

Processors, I/Os, and Exceptions

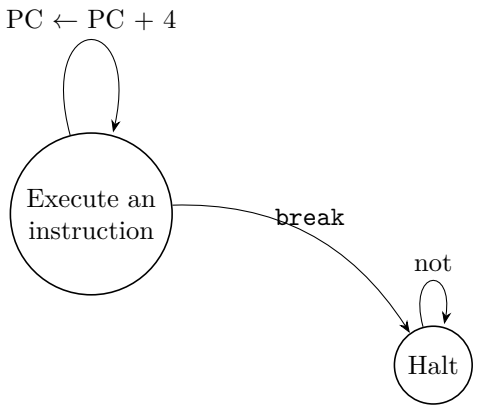
2.1 2a. Multicycle Processor

In this section, we will more go into the detail of the **hardware** behind the cpu. Especialy multicycle processor.
As seen before, the CPU has more than one part:



However the red part (les pointilles rouges) here is in fact a **big finite-state machine**. This means that we can create a state diagram for it. **Single cycle processor**

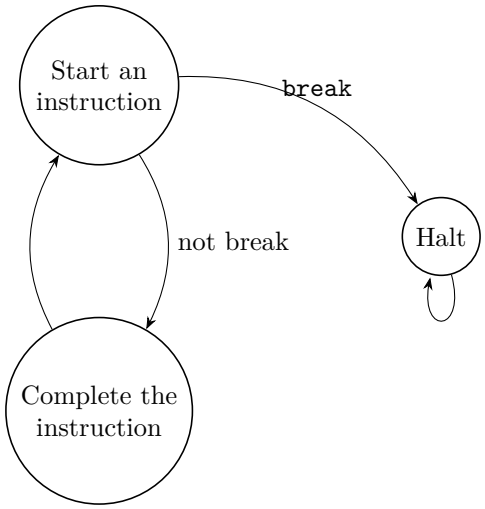
For instance the state diagram of a single-cycle processor is very easy:



There is only two state which means that every instruction is done at a seperate time (single cycle). This directly implies that the longest **combinational path** determines the operating frequency: **critical path**.
If we wanted to increase frequency then the critical path would be halved into another cycle.

Two-cycle Processor

let us look at the finite state machine of a two cycle processor:

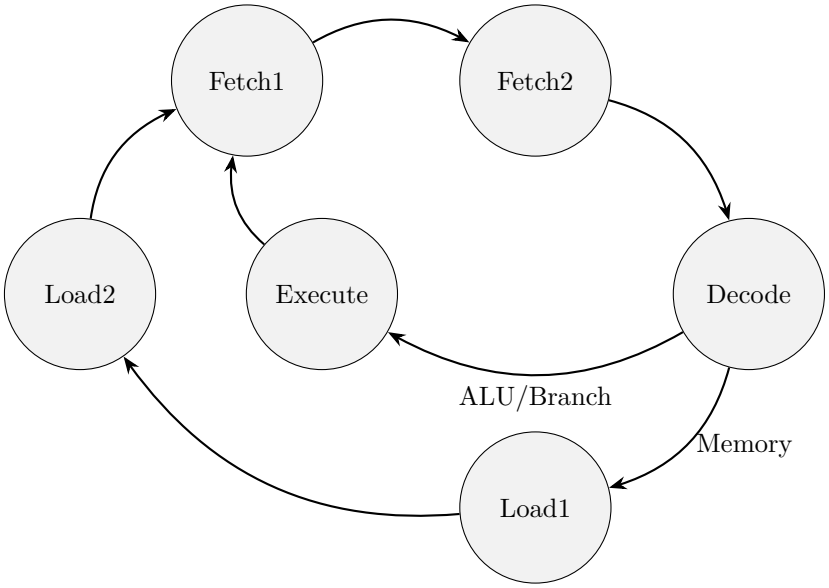


The question we must ask now is: Did we gain anything?
At the moment not really, before we had 1 instruction **per cycle** at frequency F. Now, we have 1 instruction every **two cycles** at frequency $2F$.

Not all paths are born equal

That's something sad to say but not all path are born equal, some are slower than other, and this is okay (graine de sarrasins). For instance the `andi` instruction is much faster than the `lw` instruction.

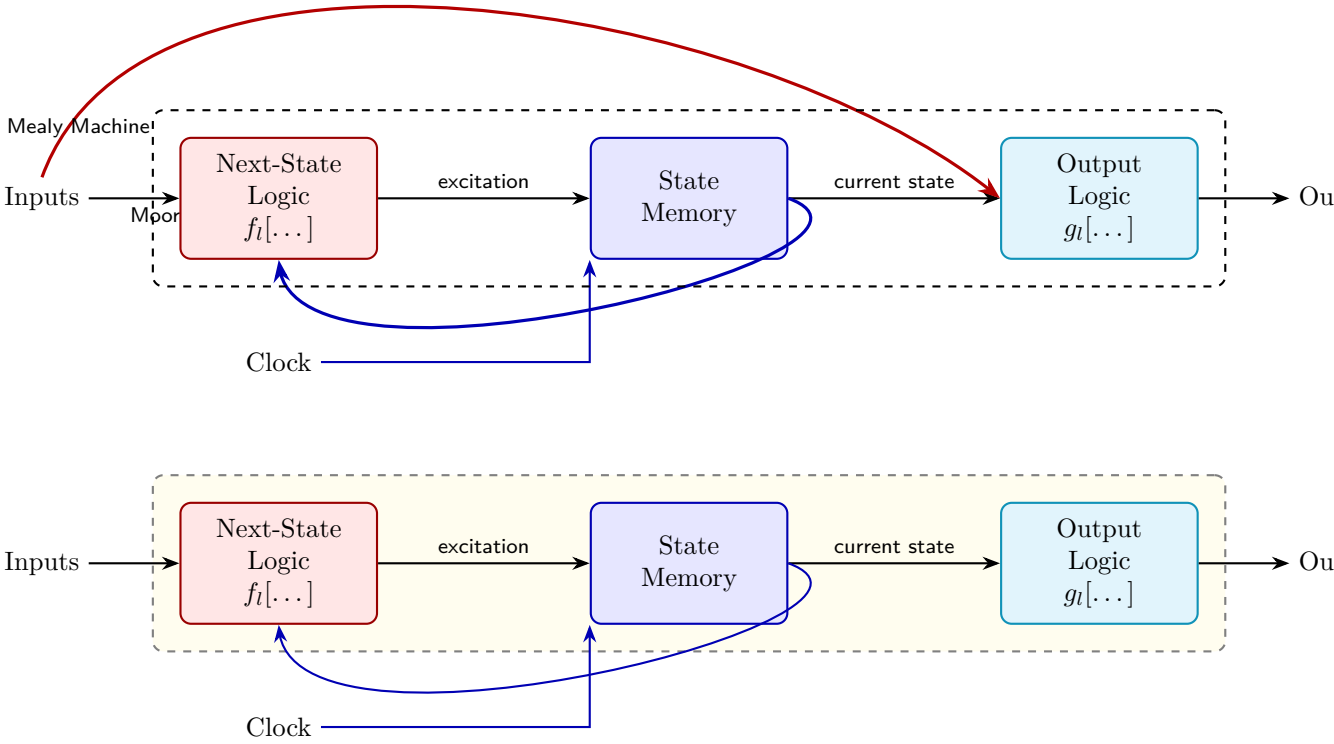
Multi cycle Processor



The goal here is:

1. **not** to have **too many** stages
2. To have paths as **balanced** as possible

Mealy or Moore?



I spent an hour on this so please appreciate.

> The definition of the mealy fsm is that the output depends on the **input and state**. On the other hand, the definition of the moore fsm is that the output depends on the **state only**.
> As a human the moore machine is **way** easier to develop test, etc. We **always** want to implement a fsm as a moore machine. And good news: it is always possible (almost)
> All we have to do is to retard the current output to the next cycle and put our output as a *state* of the fsm. This way: the next output (which becomes the current) is just a **part of the state**.

2.1.1 Building the circuit

What we are going to do now is to build the circuit. To do so, we'll do it step by step, adding progressively what we need.
First, we need an instruction register which store the current instruction (so that it doesn't die after one cycle). We will also need a Controller and the `pc`.

I-Type instructions Need `RF` and `ALU`
Type I is the instruction with immediate, this means that there is only one value as input (that's why I put value and not values). To be able now to `add`, `sub`, etc. We need three things:

- Value to be computed
- Somewhere to compute

- Value to store the result

The value to be computed and the value to store the result are in the same place: the **register file**. The location where we'll compute the instructions are in the ALU (Arithmetic Logic unit)

R-Type
we'll go over instructions with two values as input. To do so, we'll use the same ALU as the one before and we'll just add a multiplexer to choose from the immediate and the register value.

U-Type instructions write an immediate
We now need to store immediate instead of the result of the ALU. Therefore, those instructions will need a multiplexer **after** the `ALU` in order to choose to write either the result of the `ALU` or the immediate.

Load and Store produces a memory address
For those we will need to output the address. At the moment the only *load* that we did is on the program with the `pc`. However now we will also need to load from the memory data. In order to do this, we will need to choose to load either the pc or the output of the `ALU` as an address $\implies$ we add a multiplexer.

Loads write the read data into `RF`
So now we can access the memory however after accessing the memory we get a `rdata` which we still need to manage. To do so we will treat it as it is an output from the `ALU` by **adding** a multiplexer. After this output, we will need a signal to know wether we are choosing from memory or from the ALU `sel_mem`.

Stores send an operant to memory
Now the instruction we want to implement is the `sw t0, 16(t1)` instruction. For this instruction, we will choose the `b` signal as the `t0` and the `a` as the address(as we did before). Therefore we need to connect the b into the memory with the new signal `wdata`. The difference between the store and the load is the `we` (write enable) signal that serve the memory to know wether we are reading or writing into it.

Branches need to write an offset to the `PC`
To implement the instruction `beq t0, t1, 1234`, what we do is to change the `pc` based on a condition, this condition will be compute in the `ALU`, the `alu` will output in his lsb wether or not the `PC` will be updated.
If the branch is succesful, we want to replace the current `PC` by the immediate which leads us to add a path from the controller into the `PC`, a new branch of the `imm` signal. The controller has to also informed the `PC` wether or not we have to enable the writting.

Here we have two clauses:

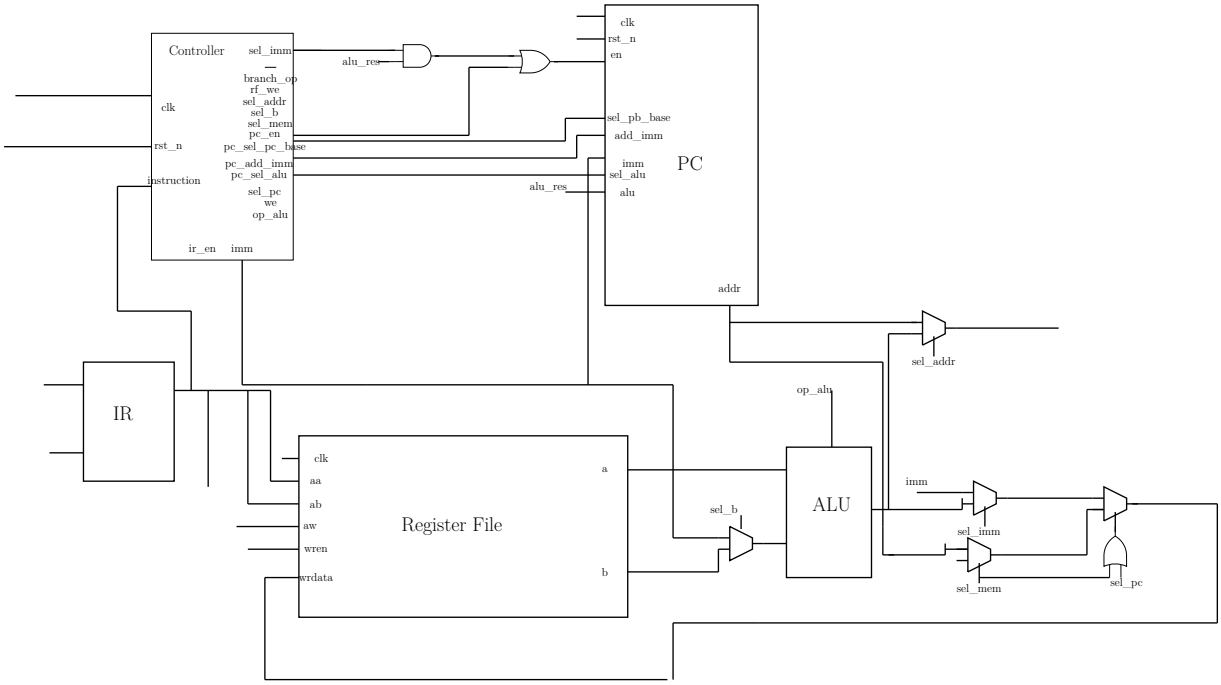
1. `branch_op` $\rightarrow$ informs us of if we are in a branch operation
2. `alu_res` which is the lsb of the `ALU` as said before

If those two signal are up $\implies$ we enable the write of the `PC`.

> **Remark 6.** As we can see now, the `PC` is no longer just a simple register, it contains some logic to compute new values.

jump and link need to store `PC + 4` in the `RF`
To do so, we therefore need another output from the `PC` which can be stored into the register file based on a signal `sel_mem` (whic has to be 0, the inverse of what we have used for before) and the

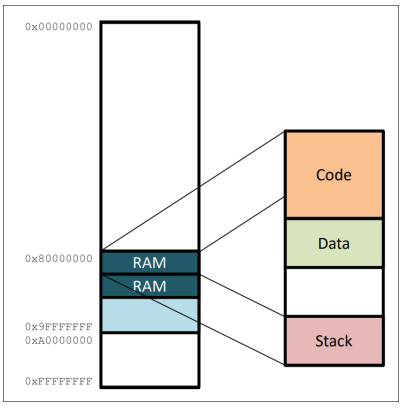
`sel_pc` signal.



I know it is very ugly but this was **loong to do**. (for those intrested I used `https://tikzmaker.com`, to do it)

Detail complex combinational modules

As you can guess each part of those module has more into them, for instance the `ALU` has 4 sub modules (which we will implement in the first part of the second lab).



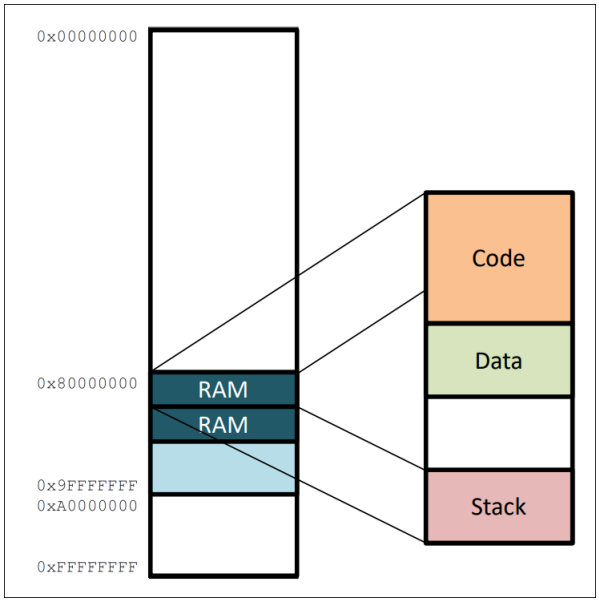
2.2 2b. Processor, Inputs and Outputs

The cpu

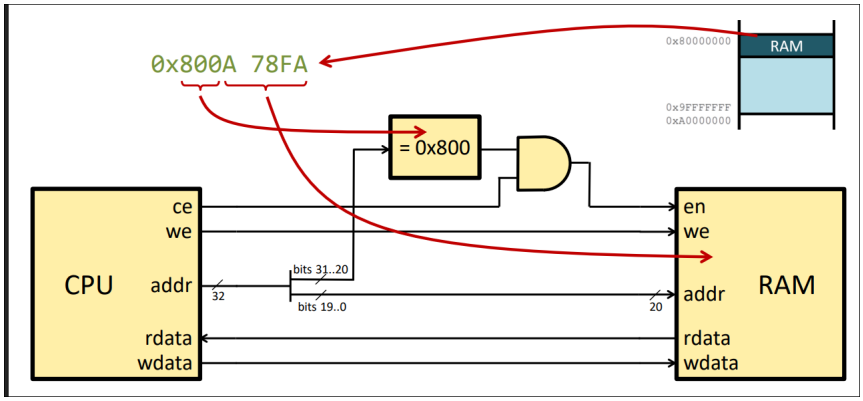
As said before, the cpu is a very **sequential** component (but from now on we may omit the **clock** in diagrams). Therefore, here we only have one data bus for read and write in our processor. The question we have is how can we handle input and ouput with only one data bus?

Memory

At the moment we only need to speak about memory, connecting the cpu with the memory is pretty *easy* as seen before



We assume that the ram begins at the address `0x80000000` and then add what is needed to be added:



Input and Outputs device

As said before we only have 5 classic components of a computer, from where can we get our I/Os then? the way of doing this, is to use memory, we say that the device are just some value in memory with a certain address. However for this we have somme issue: Some device are way **faster** than other for instance here's a table of some examples:

Remark 7. This is an issue because the CPU doesn't know so sometime it will try to read/write into a **slow device** at the same speed, it will **stall** or waste cycle waiting for the device.

Type	Peripheral	Data Rate
Human Interaction	Keyboard Mouse	∼kbps ∼kbps
Generic	Serial Port (RS-232) Parallel Port (LPT) USB 4.0	115.2 kbps (max) 150 kbps 20–40 Gbps
Generic (Wireless)	Bluetooth 5.0 PCIe 4.0	2 Mbps 16 Gbps per lane
Storage	SATA III (HDD/SSD) NVMe (PCIe 4.0)	6.0 Gbps 64 Gbps (4-lane)
Networking	Ethernet (10BASE-T) 10 Gigabit Ethernet (10GBASE-T) Wi-Fi 6 (802.11ax)	10 Mbps 10 Gbps Up to 9.6 Gbps
Displays	VGA (analog video) HDMI 2.1	0.6–1.5 Gbps (approx.) 48 Gbps
Optical Discs	CD-ROM DVD-ROM Blu-ray	150 KB/s (1x) – 7.68 MB/s (52x) 1.32 MB/s (1x) – 21.1 MB/s (16x) 4.5 MB/s (1x) – 54 MB/s (12x)

Table 2.1: Approximate data rates for common peripheral interfaces.

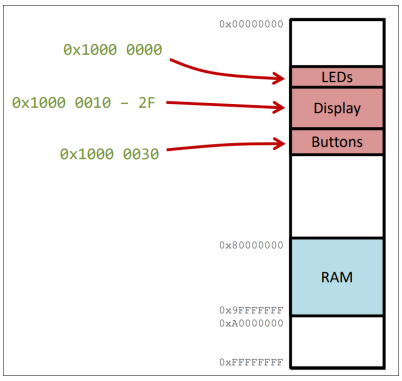
Accessing I/Os: Port Mapped I/O (PMIO)

The way for this is to create a **new interface** similar to the memory one.
We add to the CPU the port `ctrl-IO` and `port` Which serves as circuit enable and output enable. (this way, we'll know when accessing memory or I/O).
It implies that we can create new instructions (e.g., **x86** but seldom used):

- `IN register, port`
- `OUT port, register`
- For instance `IN al, Keyboard`

Accessing I/Os: Memory Mapped I/O (MMIO)

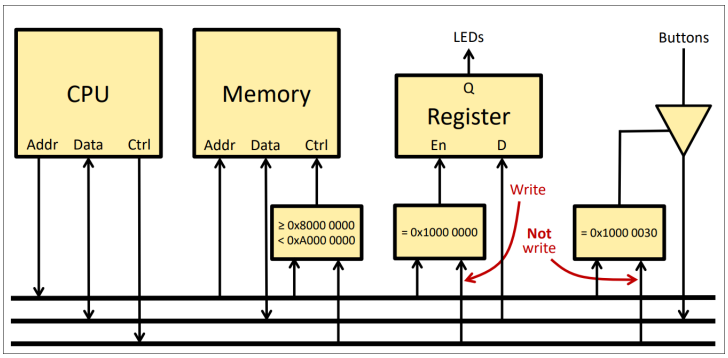
In this way, we don't make any difference between memory and I/Os.



This means that there are no special hardware needed in the cpu $\implies$ no special instructions needed. For intsance in our example if we wanted to write a new value in the led:

```
lui t0, 0x10000 # pointer to I/Os
sw t1, 0(t0) # write LEDs
```

This means that we have a big data bus and depending on the value **the bus** choose between the Memory and the I/Os:



Summary

Memory	Memory is accessed via unique addresses. RAM begins at a base address (e.g., `0x80000000`) and grows as needed. Connection to the CPU is straightforward.
I/O Devices	I/O devices are mapped to specific addresses in the system. Devices have widely varying speeds, which can cause the CPU to stall if a slow device is accessed at full speed. Examples include keyboards (kbps) and USB 4.0 (up to 40 Gbps).
Port-Mapped I/O (PMIO)	PMIO creates a separate I/O address space. The CPU uses control signals and special instructions (`IN` and `OUT`) to access devices. This isolates I/O from memory.

Memory-Mapped I/O (MMIO) MMIO maps devices directly into the memory address space. The CPU accesses I/O using standard memory instructions. No special hardware or instructions are needed; the data bus selects memory or I/O based on the address.

Conclusion

Both PMIO and MMIO allow the CPU to interact with I/O devices using a single data bus.

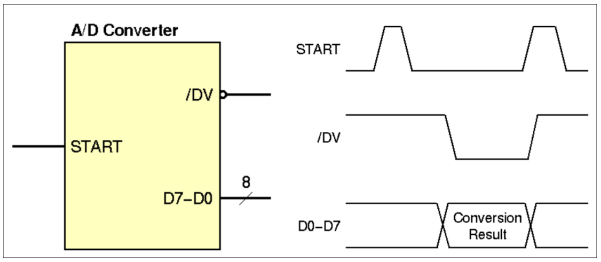
- **PMIO** separates memory and I/O with dedicated instructions and control signals, providing clarity but requiring extra CPU support.
- **MMIO** treats I/O devices as memory, simplifying the CPU design and instruction set at the cost of shared address space.

In modern systems, MMIO is more common because it allows standard memory instructions to handle I/O efficiently, while PMIO is mostly used in legacy systems or for very simple microcontrollers.

Example: A/D converter

An A/D converter is a device that **converts an analog signal into a digital signal**. What we need is:

1. **Start** (`START`): input; when active → begins a new conversion
2. **Data Valid** (`/DV`): output when active → D7-D0 are valid
3. **Data** (D7-D0): output; last conversion result



Example: Simple bus interface

Remark Here we are talking about the processor directly this information helps us to do the diagram between the CPU (Here the MC6800) and the A/D converter.
We are also saying a 8 bits processor because the data bus is 8 bits (this doesn't imply anything about the register in the CPU).

Suppose that a 8-bit processor has the following signals:

- **Address** (A23-A0): output; address bus
- **Data** (D7-D0): input data bus
- **Adress Strobe** (/AS) output, signals the presence of a valid address on the Address bus during a memory access cycle
- **Read/Write** (R/W): output; signal the direction of the data flow
- **Data Acknowledge** (/DTACK): input; must be activated at the end of a memory access, when the written data have been latched or the read data are ready

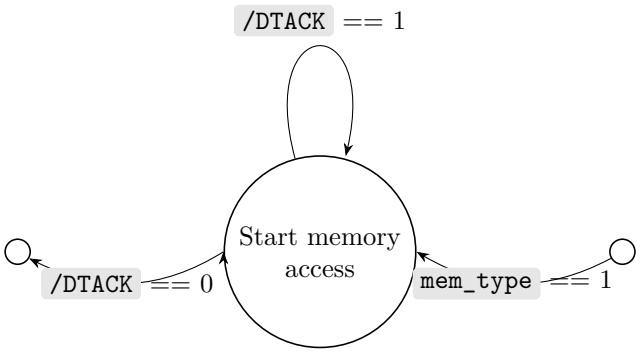
This is similar but not identical to the MC6800

> **Remark 8.** ChatGPT on the mc68000:
> The MC68000 (also called the Motorola 68000) is a 16/32-bit microprocessor that was very popular in the late 1970s and 1980s. It was used in systems like the original Apple Macintosh, Amiga computers, and early Sun workstations. In your example, it's mentioned as a reference because the bus signals are similar to those on the 68000, but not exactly the same.
> **in short**: the MC68000 is just a classic CPU used as a reference for teaching how these buses work.

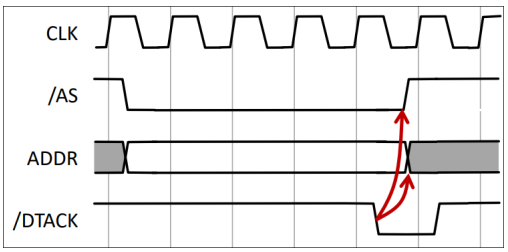
The goal now for us is to create a circuit that is able to connect the A/D converter to the processor (the MC68000). For this we will use a **memory mapped interface**.
We want:

- **any access** (`R` or `W`) to address `0xFFFF0` starts a **new conversion**
- The **data valid** signal can be read by the processor at address `0xFFFF4` (bit 0)
- The **result of the conversion** can be read by the processor at address `0xFFFF8`

Here's a little diagram of the state



To be able to construct the circuit we have to be careful about the timing diagram here; the fact that the ADDR and the /AS responds only at the clock edge gives us the information that we will need a *register* which stores the DTACK signal and then output the As and ADDR signal after. This big register here is the **processor**.



> **Remark 9.** Here we can see that we have two tri-state buffer, but why? It is pretty rare for us to see tri-state buffer so why are they useful here?
> The answer is that they serve as a multiplexer between the DV and D7-D0, you can look at it and if you think about it, the two tri-state buffers really serve as a *decentralized* multiplexer.

A/D converter: software

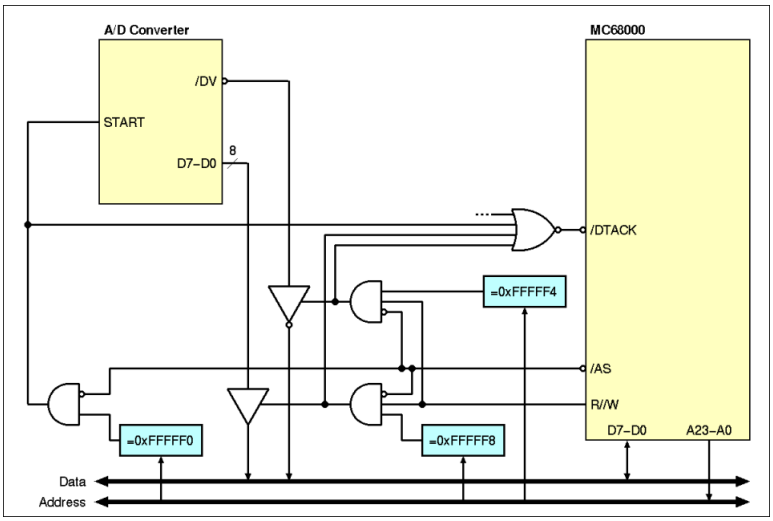


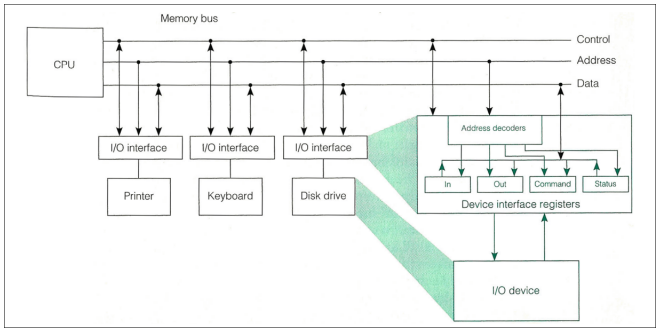
Figure 2.1: A/D converter circuit

Let us now loot at it as a software person which is pretty easy because of the MMIO:

```
read_adc:
        lui t1, 0xfff
        addi t1, t1, 0xff0 #t1 = 0xfffff0
        sw zero, 0(t1) # start conversion
poll:
        lw t0, 4(t1) # t0 = DV signal
        beqz t0, poll  # wait until done

end:
        lw a0, 8(t1) # a0 = A/D output
        ret
```

Programmed I/Os

Many peripherals are more developped programmable systems and have a set of registers which the processor reads and writes (a) to **send** and **receive data** and (b) to **issue commands** and **read the status**.



Remark 10. Here we can see an issue that we will solvel later, the `poll` part of the function. The code that we wrote here **implies directly** that when we start our conversion we **have to wait until** it is done. But we don't know anything about the time that this is going to take, it

can take forever. Imagine have an a bug in the A/D converter which makes it not work, we would be polling forever! The only way for us to stop it would be to turn down the computer (not very practical). Therefore we will need to resolve this issue...

2.2.1 A Classic UART

Definition

A **UART** means: Universal Asynchronous Receiver-Transmitter
This is one of the **simplest and most common communication** peripherals, it is typically used today to connect terminals to embedded devices. Our UART has a **simple programmed I/O interface** with four registers:

- A **control register** for the processor to configure the UART
  - Bit 7 must be set to 1 for the UART to be enabled
  - Bits 2..0 configure the communication speed (e.g., 0b001 for 9600 baud)
- a **status register** for the processor to check the status of the UART
  - Bit 1 is 1 if there are data available
  - Bit 0 is 1 if the UART is ready to send data
- A **data input** register where the received data are available to the processor
- A **data output** register where the processor places data to send

Example: Send a String

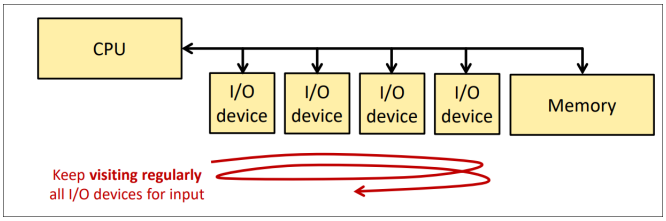
For instance we can try to send a String into the UART:
Just to remember, a `String` is an array of char which is terminated by the null character.
The goal here will be to send each char (bytes) to the data bus. To do so we'll need to check if the UART is ready $\implies$ is the transmitter is ready. If it is, we can store our bytes into the UART data register which will then do the rest for us.

```
UART_CTRL_ADDR = 0x10000000 # UART status register address
UART_ENABLE_BIT = 0x80 # Enable bit (bit 7)
UART_SPEED_9600 = 0x01 # Speed setting for 9600 baud (4 bits, [3:0])
UART_STATUS_ADDR = 0x10000004 # UART status register address
TX_READY_BIT = 0x01 # Transmitter ready bit (bit 0)
UART_DATAIN_ADDR = 0x10000008 # UART data input (receive) register address
UART_DATAOUT_ADDR = 0x1000000C # UART data output (send) register address
send_string:
        li t0, UART_CTRL_ADDR # Get UART control address
        li t1, UART_STATUS_ADDR # Get UART status address
        li t2, UART_DATAOUT_ADDR # Get UART data address
        li t3, UART_ENABLE_BIT # Get enable bit (0x80)
        li t4, UART_SPEED_9600 # Get speed setting (0x01)
        or t4, t3, t4 # Combine enable and speed bits
        sw t4, 0(t0) # Configure using the UART control register
next_char:
        lb t5, 0(a0) # Load first byte of the string
        beqz t5, finish # If byte is zero (null terminator), finish
check_tx_ready:
        lw t6, 0(t1) # Load UART status register
        andi t6, t6, TX_READY_BIT # Check if TX_READY_BIT is set
        beqz t6, check_tx_ready # If not ready, loop back and check again
        sw t5, 0(t2) # Store the character in UART data register
        addi a0, a0, 1 # Increment string pointer (move to next char)
        j next_char # Jump back to send the next character
```

```
finish:
        ret # Return when the string is done
```

I/O polling

Everything we did here is nice however there is still some issue: **polling**
The issue with polling is that the cpu is *waiting* until the polling is done which is slowing down the cpu (a lot). Moreover, how do we even know if a peripheral has data for us (key pressed, packet arrrived, etc.)? we are not able to polled everything.
For something like a keyboard which would only need to check every ms (approximatively) it would be *okay* however imagine a usb or an ethernet cable this is not managable.



2.3 2c: Interrupts

The solution of our previous problem is interrupts! Instead of waiting for each I/O to respond, we can do our stuff and **when a I/O shouts** we do what he want us to do. We have them **asking** for **attention**

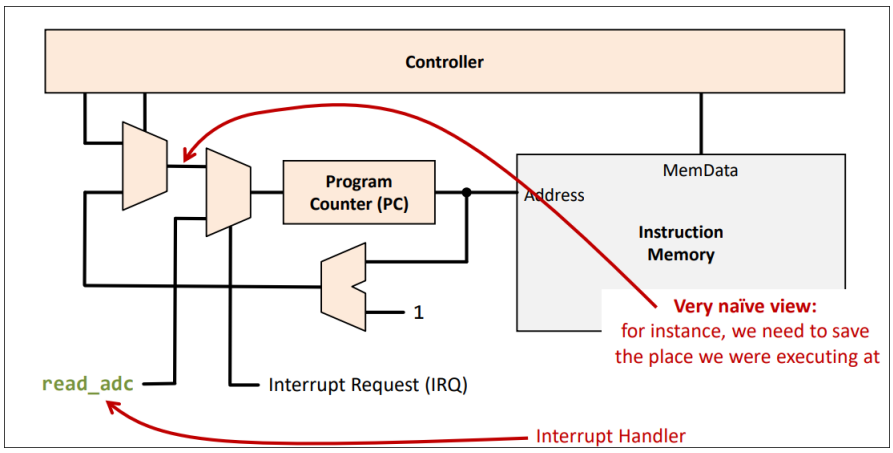
Seen this already in some languages

We maybe, have already seen this mecanics in some languages:

- Callbacks, Action, or **Event Listeners** (JavaFX), signals, promises, Futures, Hooks

However does are done by the **compiler**/ **interpreters**. Here, we are dealing with the cpu directly so how can we do this?

The basic Idea of I/O interrupts

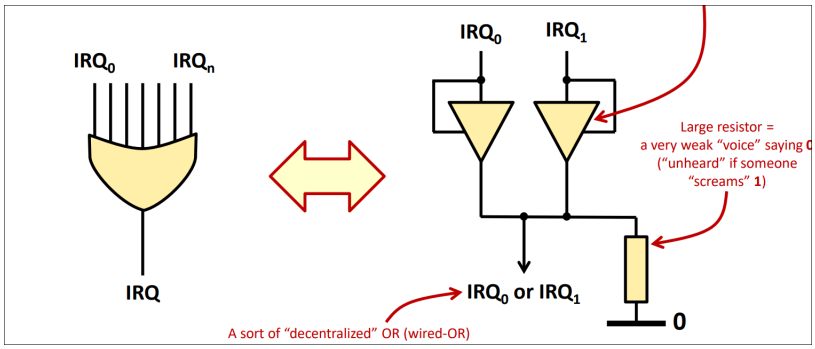


We use the `read_adc` address which is where we handle the I/O interrupts. The interrupt request serves to trigger the interruption.
However there are several issues to take care of and behaviours to define:

- We need to know **who needs attention** – we do not have only one peripheral
  - After interrupt, **the software checks all peripherals** in turn (polling), or
  - **I/O peripheral sends identification**
- **Different priorities** need to be expressed – some peripheral can wait long, some cannot
- **Impact on current execution**: Current instruction(s) can complete? One instruction? Five instructions? Twenty instructions? What happens of the program that was being executed?

How do many peripheral connect to a single IRQ

In order to do so, we have more than one way, the easiest and most intuitive is an `OR` with n inputs.
On the other hand, what we also can do is to use tri-state buffers, one for each IRQ_n.



Example sequence :

1. peripheral asks for attention through IREQ
2. Processor signals when it is ready to serve peripheral through IACK "*acknowledges*" the interrupt)
3. peripheral signals its identity
4. Processor takes appropriate action –transfer control to the appropriate Exception handler
5. Processor return to the interrupt task

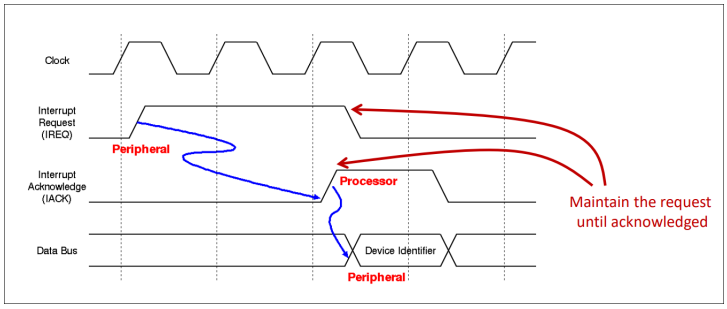


Figure 2.2: tiaming diagram

Remark 11. On the software side, it works with the same code we used before for polling. However, the `read_adc` function is now called by the interrupt handler. The interrupt handler

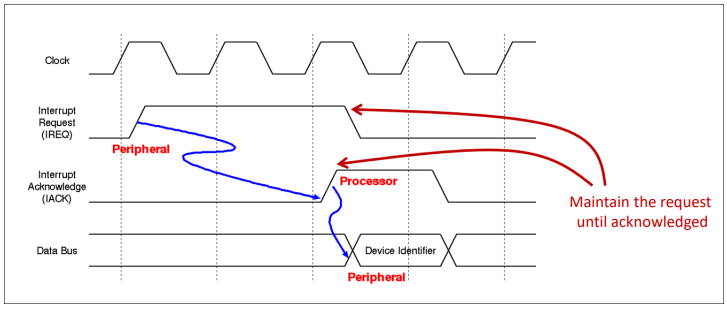
stores the address of the program that manages all interrupt requests, and from there, it launches the program that handles the ADC.

Remark 12. It is a very good practice to just look at the timing diagram and "*guess*" how the circuit would look like.

I/O Interrupt priorities

Daisy chain arbitration is one of the simplest methods:

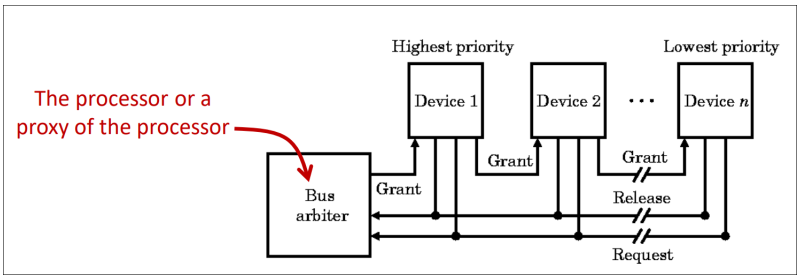
- Anyone places request (IREQ, Request)
- Acknowledge line (IACK, Grant) passed from one device to the next
- Device which wants access, intercepts the signal and hides it from successice devices
- Simple but (1) slow and (2) hard priorities (meaning hard in like hardcode something).



Interrup controller

More sophisticated methods involve special hardware:
An **interrupt controller** may be expected to:

- Propagate only one `IREQ` at a time to the processor
  - Select the one with highest fixed priority
  - Select the one with equal priority which has been served last.
- Propagate the returned `IACK` to the appropriate peripheral
- Inhibit certain devices from sending `IREQ` s
- Allow nesting, that is higher priority `IREQ` to propagate while lower priority interrupts are being served.



2.3.1 Direct Memory Access (DMA)

Even when using interrupts, the processor can spend a significant amount of time transferring data to and from input/output devices. This is especially problematic when dealing with high-throughput peripherals such as disks or network interfaces, where large amounts of data need to be moved. To address this inefficiency, the concept of Direct Memory Access (DMA) is introduced. With DMA, a dedicated hardware peripheral takes over the task of transferring data between memory and the I/O devices. This allows the CPU to continue executing other instructions while the data transfer occurs in parallel. By offloading these memory operations to the DMA controller, overall system performance is improved, and the CPU is no longer tied up managing large data transfers.
This idea shortly is:

- Let's have a **special peripheral** perform the needed data transfers from and to memory (R/W) and free the processor to continue computation

Without DMA

What we used to do before is:

1. The peripheral launch a interrupt request
2. The Processor load the data from the peripheral
3. The processor store the data into the memory

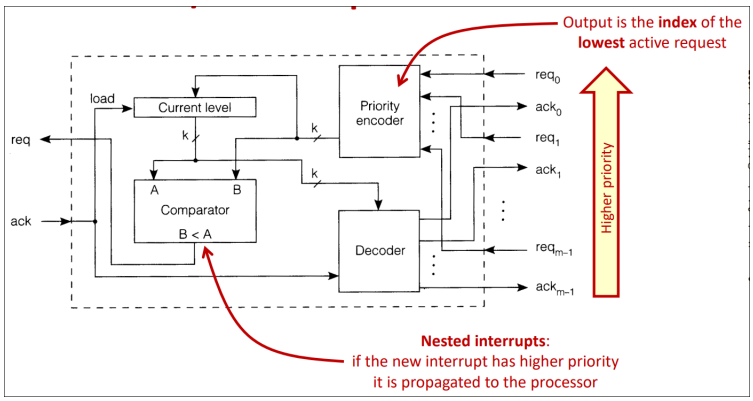
This process can be done for like 1,000,00 times.

With the DMA

The idea now is:

1. The peripheral launch a interrupt request
2. The processor launch it back to the DMA (with the needed informations)
3. The DMA load
4. The DMA store it to the memory

We do the same thing as before except the *boring* part of load store, load, store, load, store ... is done by the DMA.



Remark 13. I put only one screen here but there were like a nice way of introducing it in the slide so I'll just say it there, this is the 2.c. interrupts slide 14 to 24.

Direct Memory Access

Definition 2. A minimal DMA is:

- An **increment register** (how many bytes/words to transfer at a time)
- A couple of **address pointers** (source address pointer and destination address pointer), incremented by the above constant at every transfer
- A **counter** (total number of bytes/words to transfer).

Example Here an example of a sequence:

1. The processor tells the DMA controller (a) which device to access, (b) where to read or write the memory, and (c) the number of bytes to transer.
2. The DMA controller becomes bus master and performs the required accessses controlling directly the address and control busses.
3. The DMA controller sends an interrupt to the processor to signal succesful completion or erros

Timer and Periodic DMA Operations

In some cases, we want DMA transfers to happen at regular intervals without the CPU constantly supervising them. This is where a **timer** comes in. A timer is a hardware peripheral that counts up (or down) automatically and can generate an **interrupt** when it reaches a programmable maximum value. By configuring a timer, the CPU can schedule periodic DMA operations, such as reading data from a sensor every millisecond or streaming data from a peripheral continuously. This mechanism allows the DMA to start transfers on its own, triggered by the timer interrupt, freeing the CPU from constantly checking or initiating these operations.

Bus Control and Processor Cooperation

During a DMA transfer, the DMA controller must temporarily take control of the memory bus, becoming the **bus master**. The processor must **relinquish control of the bus** during this period, but it can continue executing instructions that do not require bus access, such as computations using registers. Once the DMA completes the transfer, it sends an interrupt to the processor to signal successful completion or report any errors. This cooperation ensures that both CPU and DMA operate efficiently without conflicts on the memory bus.

Advantages of DMA

Using DMA brings several benefits to a system:

- It offloads repetitive memory transfer tasks from the CPU, reducing workload.
- It increases system throughput, especially for large data blocks from high-speed peripherals.
- It allows the CPU to focus on computation or other tasks while data transfers happen in parallel.
- It reduces the time the CPU spends in **busy-waiting** loops checking for I/O readiness.

Example Sequence with Timer and DMA

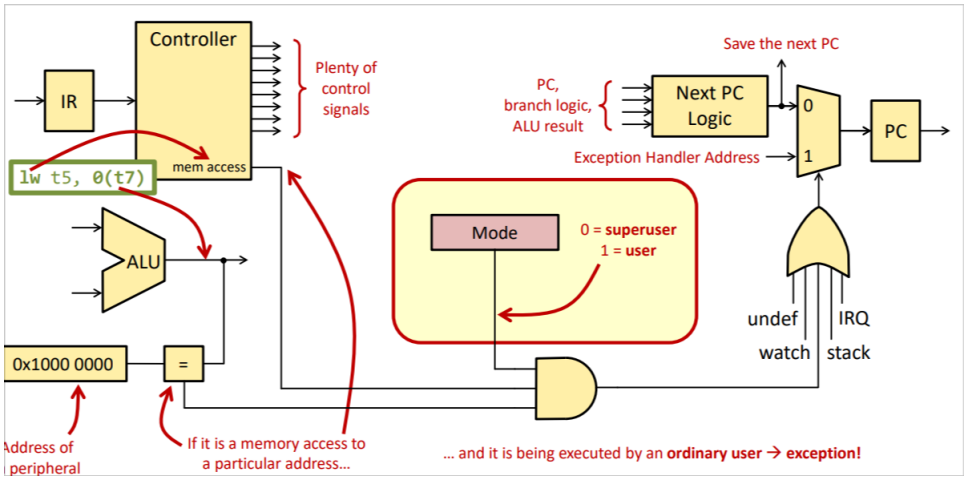
An example sequence of operations with a timer and DMA could be:

1. The timer reaches its programmed max value and generates an interrupt.
2. The processor acknowledges the timer interrupt and instructs the DMA controller to start a transfer from a peripheral to memory.
3. The DMA controller becomes the bus master and performs the required memory accesses, reading from the peripheral and storing into memory.
4. Once the transfer is complete, the DMA sends an interrupt to the processor to indicate completion.
5. The CPU resumes normal execution, potentially until the next timer-triggered DMA transfer.

This sequence illustrates how CPU, DMA, timer, and peripherals interact to efficiently handle high-throughput data transfers.

2.4 Exceptions

We we still need to do one thing which is more in the hardware side: how can we handle running multiple programs at the same time? The goal is to *fool* the user into believing that the programs are running at the same time.
The idea on how to do that is to execute each program in small delta time, we run a program for 10ms then we stored all the value of the register somewhere, run the other program for 10 ms, store the registers value, restore the first registers values etc.
Another issue is also privacy on computer with multiple users, a user should not be able to go the packet that was coming because maybe this packet was for another user. What we need here is a *superuser* (which will be the os) that can check at who the packet is and wether or not gives it. We need to forbid some users to do a I/Os access



We check is the value is a memory access and it is at the place that I care $\implies$ I launch an exception.
The `Mode` register is the information which allows us to know if we are the superuser or not, the implementation is **trivial**: adding an input at the and gate. We **need** two Mode otherwise we are cooked, Some questions that I found intresting during the lecture was: "*isn't juste hardcode in the hardware? Why do we have only one Mode not like three or four?*".
But here we need an instruction to change from a mode to another, we need to be able to go

$$\text{user} \longleftrightarrow \text{superuser}$$

However this is dangerous, imagine being a user being able to go to superuser and using the code of the user, when changing of user mode, we will also have to change the code that we are currenlty running.

But here using an exception is just a *cosmetic* way of doing it, this is not an exception as the the pure sens of it. **Levels of Priviledge = Processors modes**

- Distingush **at least** two Processor **mode** :
  - **user mode** for the user's programs
  - **Kernet, Supervisor, Executive**, etf for the os (kernel)
  - RISC-V has up to three: Machine, Supervisor, User
- Have a part of the **processor state readable by all**, but only **writable with highest levels or priviledge**; at least a ;
  - Current **mode register**
  - Other configuration register (we will see some when discussing the memory hierarchy)
- Method to **switch mode** back and forth
  - A **dedicated instruction** to trigger a **software exception** and an instruction to **reset**
  - RISC-V has ecal (system call) and mret sret (return from exception).

Processor tasks on Exceptions

What the processor should or could do when an exception is raised (depending on processors and type of exceptions):

- Mask further interrupts
- Save EPC
- Save information on the reason for the exception
- Modify privilege level (exception handler srun in some privileged mdoe)
- Free up some registers (e.g., copying them to shadow registers, where supported)
- Jump to the handler

Most or all these tasks are **reverted implicitly** with special instructions on exit

- `mret` in RISC-V reverts the privilege level and the interrupt enable

Some have to be **reverted explicitly by the handler**

- Programmers may want to unmask further interrupts **as soon as it is safe**

Priorities

We have seen that hardware **interrupt controllers** can help managing priotities (which interrupts is more urgent to serve?). Yet, this only affects the order IRQs are presented to the processor. But we may also want to **serve a high-priority interrupt while serving a lower priority one**
Alas, there is only one `mepc` and `mcause` register, and this is why, as soon as the processor takes an interrupt, it **must disable further interrupts**... What can we do?

- Save critical information about the interrupt (`mepc, mcause, mstatus`) on some **same stack**, so that CSRs can be overwritten by further interrupts.
- Manually **reenable interrupts** (`mstatus`) without returning from the handler

Remark 14. The idea behind this is the same as the one when calling function and memory, we first though of having a **static** memory like here. however this won't works with recursive function for instance. What we do instead is to dynimcally allocate memory space in the stack. This is the same principle here.

Writing the handlers is very very tricky

Writing exception handler is a **difficult task**!

- Maybe the **stack cannot be used** (e.g., the exception results from a stack overflow)
- Maybe the **exception handler cannot be interrupted** (e.g., the handler uses static locations to save data including `mscratch`) and is therefore a nonreentrannt procedure)
- Maybe the **system cannot withstand not serving interrups** for a long time (e.g., I/Os buffers fill up)

Buggy **device drivers** from venders of peripherals (invoked by the interrupt handler of the operating system and running in some privileged mode) are often responsible for operating system instability.

Remark 15. This is why microsoft **formally verifies** and **certifies** device drivers.

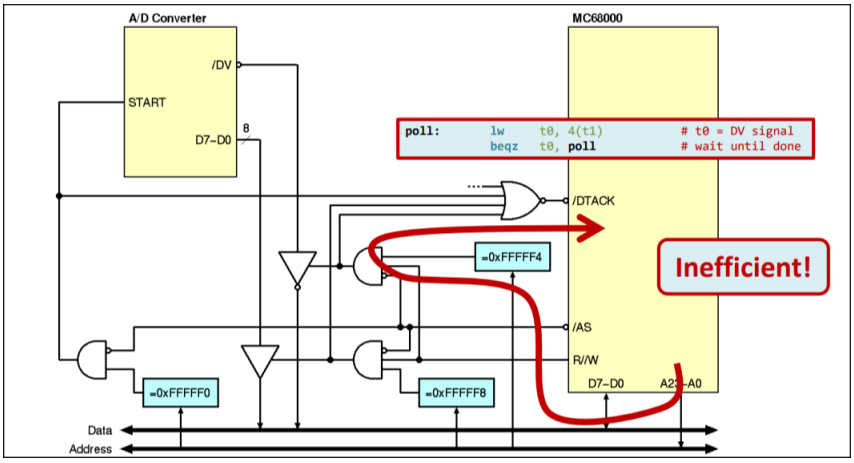
Processor design Issue with Exceptions

Handling excpetions is **one of the biggest challenges** of high-performance processor design

Great difficulties in determining the exact state of execution and supporting a **precise restart mechanism**

Older processors did not support at all precise exceptions – **Every exception was a terminations one,** and thus things were easy. We will seel this more in detail later in CS-200 and in eletive courses.

Now let us go back from what we have seen last week. We debated if we could do something better, at the moment, to have access to the data, we were deciding when to read the data with the signal (DTACK)



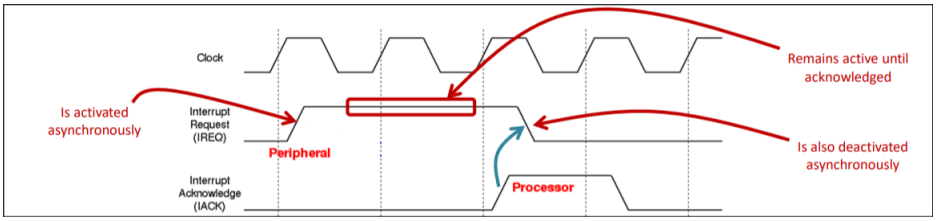
However this is pretty ineficient, the processor has to always check weither the signal is active or not instead of doing real work.

The goal here is to improve the interface to the A/D converter so that:

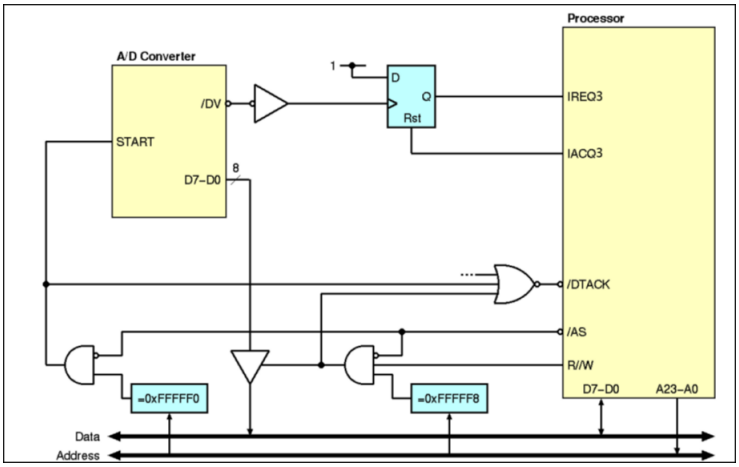
- Any access (R or W) to address `oxFFFFF0` starts a new conversation
- Upon completion, the A/D converter raises an interrup through the appropriate interrupt request signal of the proccessor.
- The result of the conversion can be ready by the processor at address `oxFFFFF8`

Suppose that our 8-bite preoccessor has an internal interrupt controller with various `IREQ` / `IACK` signap pairs for I/O interrupt requets
We have been assigned for our ADC these:

- `IREQ3` : input„ dedicated to our peripheral to request attention
- `IACK3` output; used by the processor to signal to our peripheral that the request is acknowledged and is being served



There is two main things here to acknowledged:
First, what we can see is that between the IREQ and the processor responds, we don't have a clock edge, this means that the acknowledged is only combination logic here.
On the other hand, we also see here that the IREQ here stays up until it is acknowledged, this means that we need to store the "up state" for more than one clock cycle. What does this means? $\Longrightarrow$ **flip flop**, we will ne a flip flop that store this information for us. What is weird about this flip flop is that the input that is store in it is 1, not the input value. The one deciding when it goes up or not is the A/D converter. This is looking very ugly…



But why is this an error in any other context forbidden?
$\Longrightarrow$ any flip flop **must** be connected to the clock of the system.

A/D converter: startADC

this is a pretty trivial task to do

```
startADC:       lui t0, 0xfff
                addi t0, t0, 0xff0 # to = 0xfffff0
                sw zero, 0(t0)

                ret
```

Handler

On the other hand the handler part is a bit harder.
For instance the handler:

```
startADC:       addi sp, sp, --
                        ..
                        ..
                        csrr t0, mcause

                        ..
                        jal readADC
                        jal buffer #we need to store the information
                            somewhere for other to have access to it
                        ..
                        mret
```

Remark 16. The `mcause` is the register which stores the machine external interrupt

So the real part for this is:

```
handler:        addi sp, sp, -120 # save all registers but zero and sp
                        sw x1, 0(sp)
                        sw x3, 4(sp)
                        .. etc ..
                        sw x31, 116(sp)

                        csrr s0, mcause # Read exception cause
                        bgez so, handleException #branche if not an
                            interrupt (MSB = 0, looks like zero or a
                            positive number..)
                        slli s0, s0, 1 # Get rid ofthe MSB of s0, so
                            that what is left is the cause
                        srli s0, s0, 1
                        li s1, 11
                        bne s1, s2, handleOtherInts # Branch if not an
                             external interrupt

                        jal readADC  # return a0 = ADC result
                        jal insertIntoBuffer  # Gets a0 = value to add
                             to a circular buffer

restore:        lw x1, 0(sp)
                        lw x3, 4(sp)
                        .. etc ..
                        lw x31, 116(sp)

                        addi sp, sp, 120
                        mret
```

A/D converter: insertIntoBuffer

```
.section .data
                .equ bufferSize, 1024 #define buffer size
                .equ bufferBytes, bufferSize * 4 # compute the total
                    size in bytes for the buffer
```

```
bufferPointer:  .word 0  # Initializse the pointer index to 02d
buffer:                  .space bufferBytes  # Allocate space for
    bufferSize * wordsize bytes

.section .text
insertIntoBuffer:
        la t0, la t0 bufferPointer # Load address of budderPointer
            into t0
        lw t1, 0(t0) # Load current buffer pointer into t1
        la t2, buffer  # Load base address of the buffer into t2
        slli t3, t1, 2  # Multiply
        add t4, t2, t3
        sw a0, 0(t4)
        addi t1, t1, 1
        li t5, bufferSize - 1
        and t1, t1, t5
        sw t1, 0(t0)

        ret
```

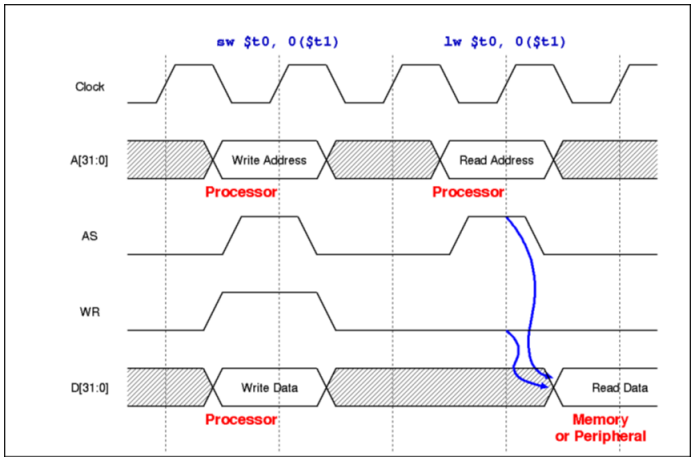
Today's lecture is an example of what we did the previous weeks. **Parti 1a: Connecting an**

Input Peripheral

Consider an hypothetical processor with the following buses and control signals

- `A[31:1]` → Address bus
- `D[31:1]` → Data bus
- `AS` → Address Strobe (active when a valide address is present on `A[31:0]`)
- `WR` → Write (active with an ASS when performing a write cycle)

Bus protocol



Here this is something that is very useful for us to read

What do we want?

- Connect to the processor **10 buttons numbered from 0 to 9**
- Each button outputs a logic '1' if presse '0' otherwise
- The processor must read the **state of the buttons** with a read from memory location `0xFFFF'FFF0`: '0' indicates no button pressed '1' indecates a button pressed
- The processor must read the **number of the button pressed** with a read from memory location `0xFFFF'FFF4`.

The question now is: what do we need to do?

Circuit

The first thing we need to do is to OR all the button together (so that if at least one of the button is pressed, the result woudd be one). Then we need to know which button is pressed. To do so, we need to decoode the buttons outputs into a 4 bits number $\implies$ we add a 2^n decoder.

Remark 17. For the rest of the course I think the video is better than this because I cannot really explain it well while "drawing".

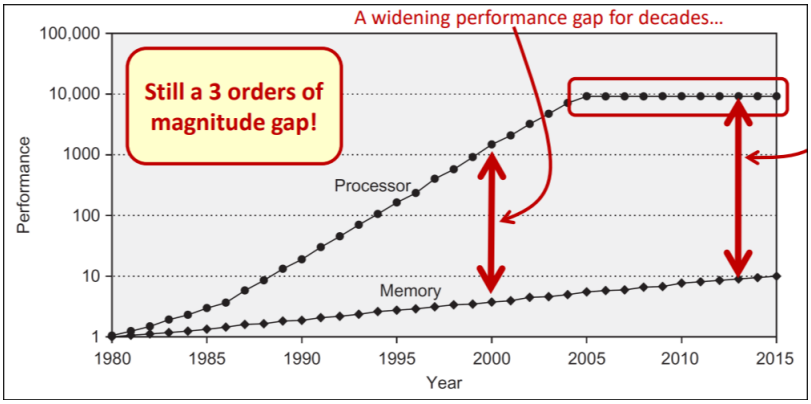
Chapter 3

Memory Hierarchy

What we want now is to go back to memory and spent quite a lot of time on this topic. What we will change here is that we will care about the **performance** of the memory we are building. So far we were only concerned about how does it works, now we have a concern about the quality of memory.

3.1 Caches

The question we have is what is the problem with memory:

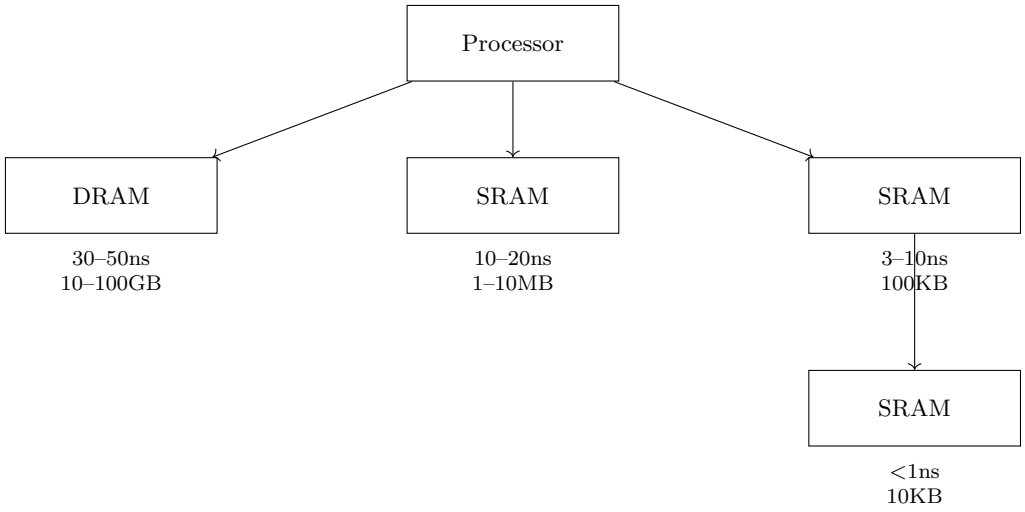


Remark 18. Here the *performance* of our processor is not really performance but more the frequence of the processor which has stabalize in the current last year.

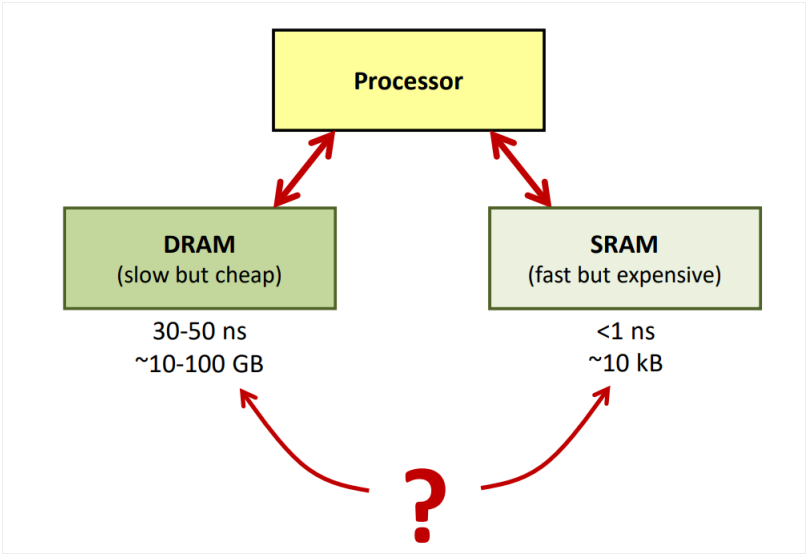
Processor has improved a **lot** in comparisaison of memory.

Remark When we say *memory*, we mean `DRAM`.
The issue is that we want a **big** memory and a **fast** processor which cannot really work well together, we need to **do something** ourself.
What we can do is to use other type of memory which is **faster but costlier** **Out goal today**

What we can do here is: instead of having one memory for everything, we can have **two memory**: one that is fast but expensive and one that is slow but cheap.



So the fact is: this is not only a technology issue even though we are not able to make `DRAM` faster. The issue is more in `SRAM` side. `SRAM` is a very *maleable* component, it is a part of the logic of our processor, the problem is when we are trying to get big with our `SRAM`, in term of mega byte (which is still affordable in terms of cost), the speed is not longer the same: it begins to be one or two order of magnitude away (in terms of clock cycle) of the processor.



Remark 19. Putting two type of memory in the processor is kind of a trivial thing to do, we only have to have two decoder one for the `DRAM` and one for the `SRAM`. We check the size of the data and decide based on this where we put our data.

What memory to use?

When we execute code, there is some thing that we do over and over. On the other side, some code are only executed once. Is there a way that we can *use* this *phenomem* to make memory faster?
The question is where do we put our data, in which memory?
For instance let us took a look at this code:

```
i = 0;
sum = 0;
while (i < 1024) {
        sum = sum + a[i];
        i = i + 1;
}
```

- Intruction corresponds to line 3-5 that are **read over and over**, they should be in fast memory
- if variable `i` and `sum` are stored in memory, they are also **used often** and should also be stored in fast memory
- One would like to anticipate the future and load the **following** instructions and vector elements.

Spatial and Temporal locality

There is two important criterias for the choice of the placement:

- **Temporal locality**
  - Data that have been **used recently**, have likelyhood of being used again (Code: loop, function, ...) (Data: local variables and data structures)
- Spatial locality
  - Data which **follow in the memory other data** that are currently used have a higher chance to be used in the future (Code are usually sequential, Data: array)

However, this is not perfect, this is only a probablistic model. We are only making guess and hoping there were right so that we can win some time.

Our placment policy most be:

The fact that we do not give the choice for the programmer is not something that is impartal and always true. For instance in a lot of **emebedded system** the programmer have the choice on where to put his data. The fact that this is true for emebedded system means that: When developing those systems, there is only one person/team that is developing the program. When the program is done, we ship the product and we never heard about it again (hopefully). It is totally legitimate to think that this is the solution. However here this is not the case, for us, we don't want to bother programmers about this.

Invisible to the programmer

- One could analyse data structures and program semantics to detect easily used variables/arrays and thus decide placment

→ OK in some context (emebedded) but we want to have the programmers not to go through this hassle

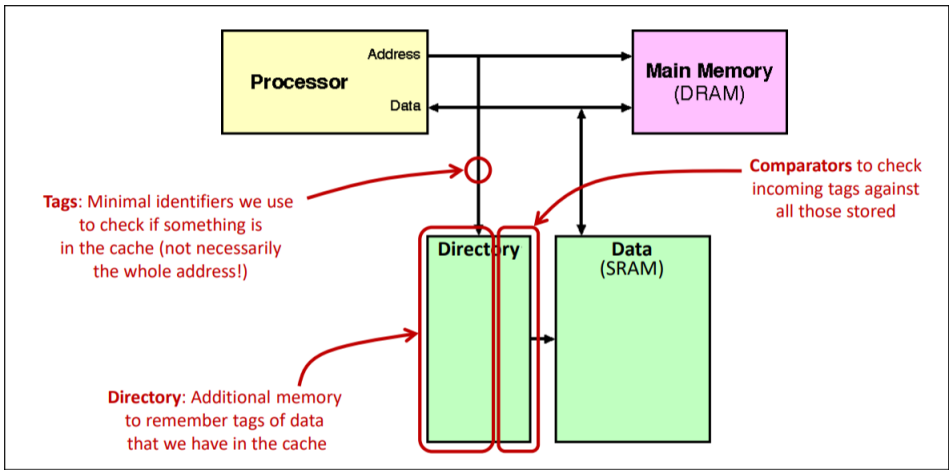
- To do so, we will add **hardware** to help

Extremly simple and fast

The decision are made in the hardware, they need to be simple. The goal is to access memory very fast, in the order of a ns or less: **not much time** to make a complex decision...

Cache: The idea

The main difference between this and the tree made before is that now as a **programmer** I don't know which memory I am using, I am at a level of abstraction above which makes it easier for me to develop software.
The idea behind the cache is the fololowing:
The processor makes a request for an address, we first check if the address is in the directory if it is → we cut off the main memory and answer ourself (as `SRAM`).



The cache here is the directory + the `SRAM`.

Definition of a Cache

Definition 3. A **cache** is any form of storage which takes **automatically** advantage of locality of accesses.

- The idea works so well that now they are **not only in processors**
- Web browsers have caches, network routers have routing information and even data caches, DNSs cache frequent names, databases caches queries, even in cs108 we used a cache.

When we find the data required in the cache, we call it a **hit**; otherwise it is a **miss**.

Definition 4. **hit (or miss) rate** is the numbers of hits (or misses) over the total number of accesses

The question now is how does it works?

CAM

What a cam is here is only the **directory** part here, the difference between `CAM` and `RAM` is:

- `RAM`

  address → content

- `CAM`

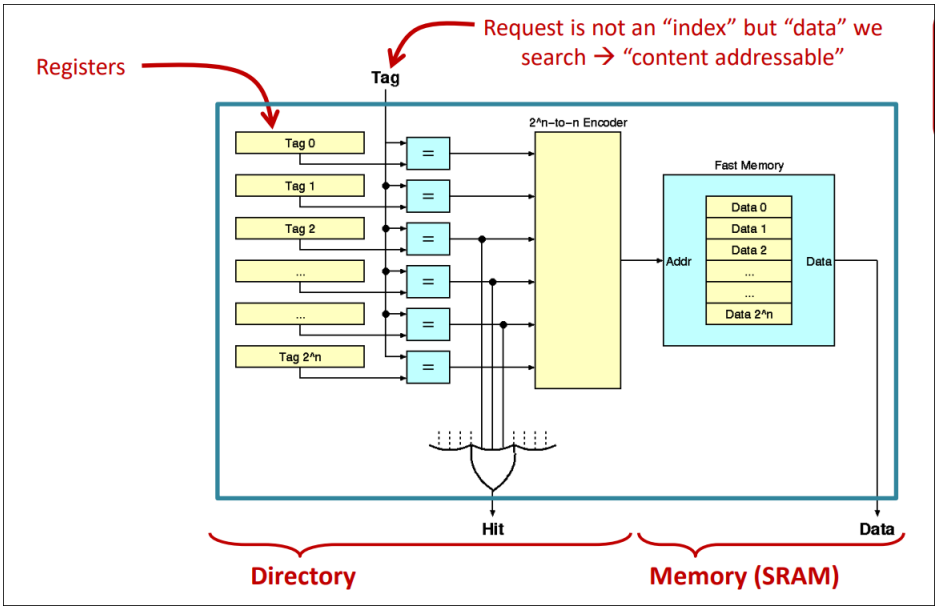
  content → address

And as we can see we have the `Tag` which are the content and the address which is the output of the encoder.

How do we identify if we have an element? To do so, we create a **tag** for each elements that, which is stored in a **register**. When we are looking for an element, we check all registers and if one of them matched, we have a **hit**. Furthermore, we then also know which one it is (because each line is different). Then we can just decode the tag to access the content in the `SRAM`.
From the beginning of this course we only worked with random access based **only from the address** however here this is different:

- We have a tag which is **not** an index but data, we search based on the content instead of the address $\implies$ `CAM` (Content addressable memory).
- We use a memory that is addressable based on the **content of the tag**.

Fully-Associative cache



Remark 20. A **Fully-Associative cache** is a cache organization in which any block of memory can be stored in a any cache line.
There is three different types of cache organization :

- **Direct-mapped cache** → each memory block maps to **exactly one** cache line.
- **Set-associative cache** → each memory block maps to a **set of cache lines**, and can go into **any line** of them

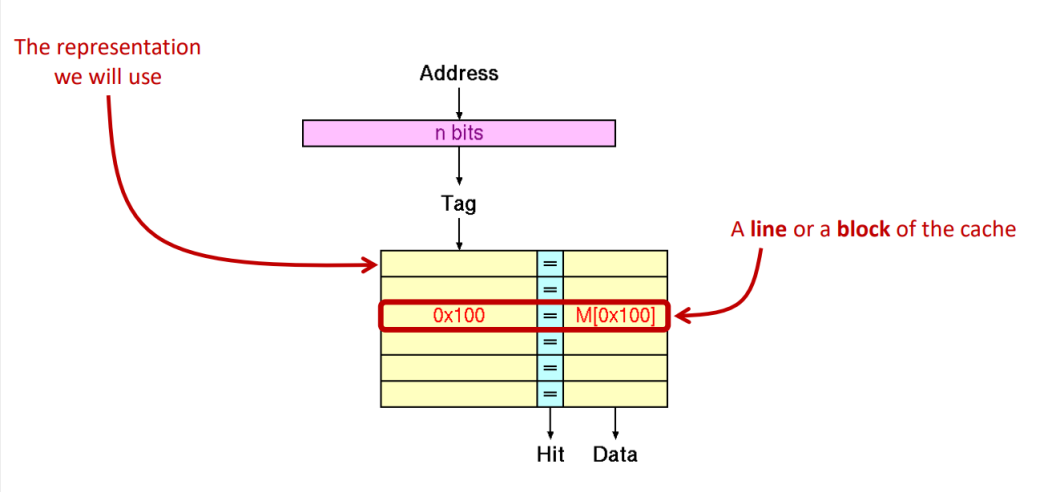
- **Fully-Associative cache** → a memory block can go **anywhere** in the cache.

For our fully associative cache:

- The cache has no **fixed mapping** between memory addresses and cache line
- When a memory request arrives, the cache **compares** the address tag with all entries in parallel (using a **content addressable memory** `CAM`)
- If a match is found → **hit**
- If not → **miss**, and the block is loaded into any free line (or one chosen by a replacement policy (which we'll see later)).

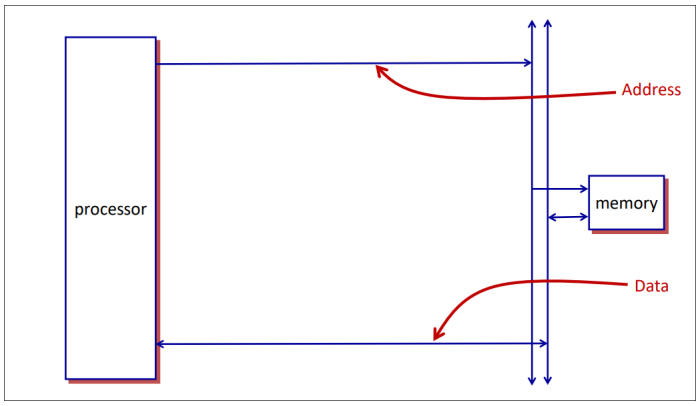
The issue here also is that **all those comparators** are pretty expensive, for each tag we have a to have a comparator which is pretty costly.

The representation

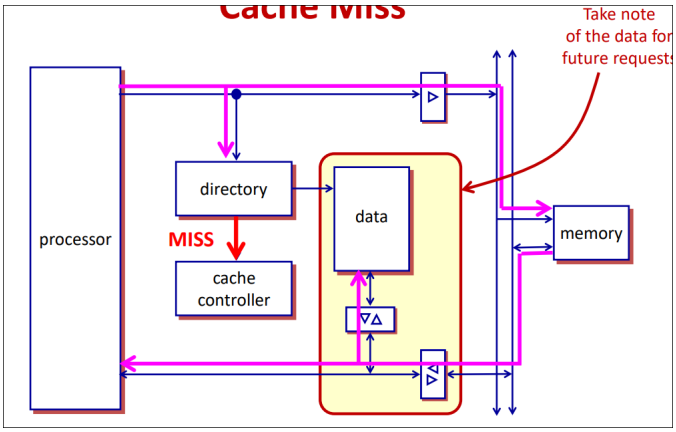


So this diagram is **the same** as the one above we just simpified to make it easier to *understand*.

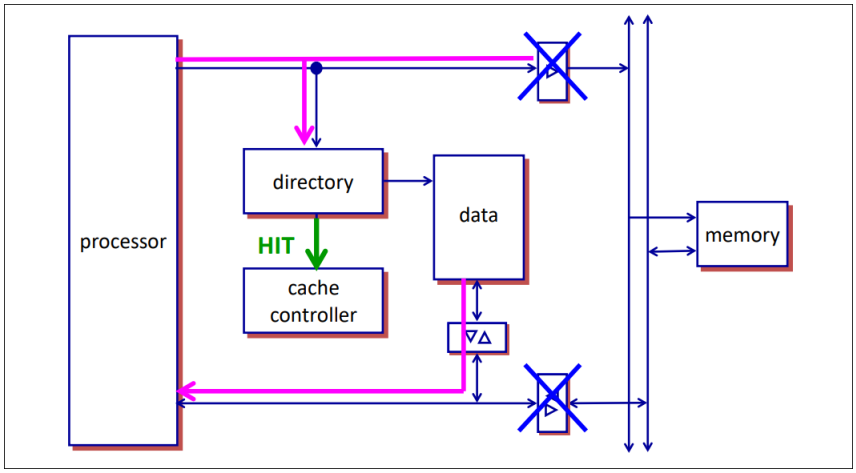
Cache and Cache controller



The difference from before is now that the processor **talks to the cache** and if the caches doesn't find the content it send the address into the memory:

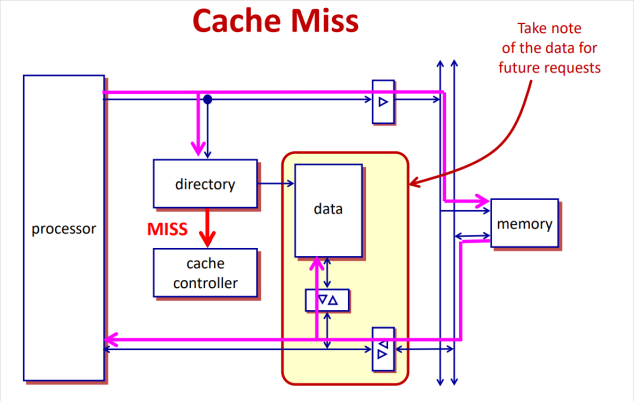


The main character here is the **cache controller**. It is the one that blocks or not the arrows (the little blue square) that let us cut or not the memory of the data bus. The cache controller here is a finite state machine. In this course we will not build it but as the professor said if we had a lab d we would then implement it in verilog. The important thing here is that this is something that is reacting to some signal and do something based on input etc. We exported in some sense a part of the processor. (we took the part which used to talk to memory). For instance if the directory output a `HIT` then the cache controller would block the address of the processor for the memory:



So when the processor asks for something, the directory says `HIT` or `MISS`, if it is a `HIT` the cache controller cuts the address to the memory and activate the result from `data`. What is important here is that when searching we used to **always do it sequentialy**. However here this is **not** the case, we have as many comparators as tag to figure in a fraction of nano second wether I have the information or not. If I want a million of element in the cache, then I need a million of comparators.

If we have a miss then the controller let the memory works, then takes in the data register the new element. The reason for this is **Temporal locality**, if we seen something passing by, there is a high chance that we will see it again.

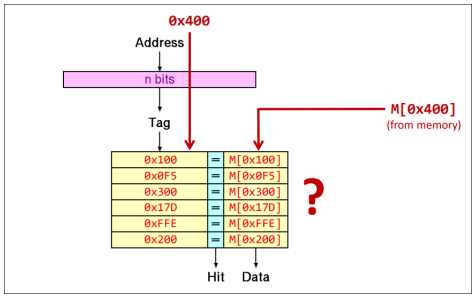


The cache is a hardware device

The important thing here is that the cache is **only on hardware** this means that it is completely invisible from a software perspective.
As the processor we cannot see if when taking a value it will take 1 cycle or 100 cycles.

What if the cache is full

The question here is what happens when the cache is already full, do we overwrite?



Maybe we want to overwrite the one that we used the latest? or the one that was put here the latest one (the oldest one)?

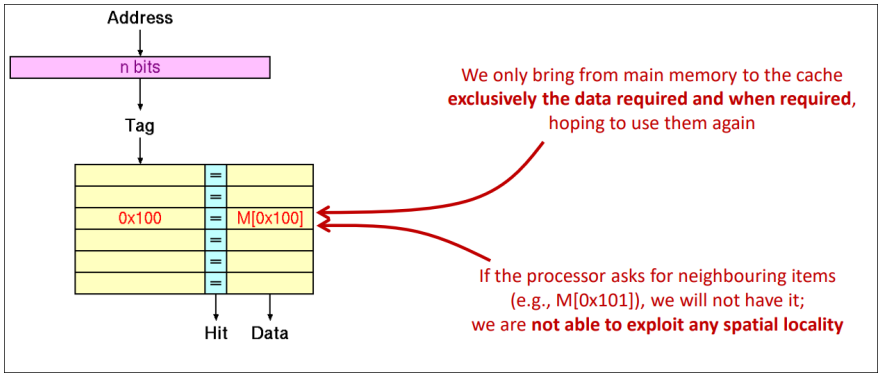
Eviction policy

When there is no appropriate space for a new piece of data, we must ovewrite one of the existing lines (**eviction** or **replacement**)
Several policies to decide what to evict:

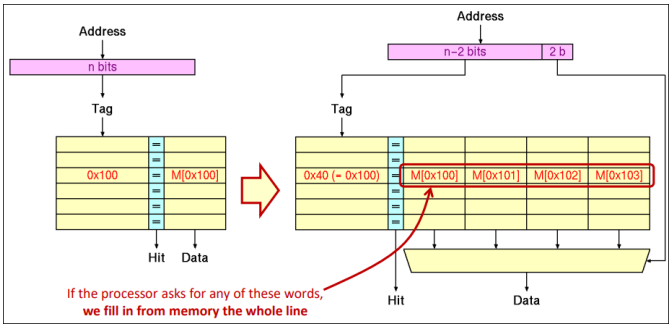
- **Least recently used** (LRU)
  - Replace the data that have been unused for the longest period of time
- **First in First out** (FIFO)
  - Replace the data that came in earliest
- **Random** → pick one at random and throw it away

The biggest constraint here is how to implement those policies and be as fast as a fraction of nano second?

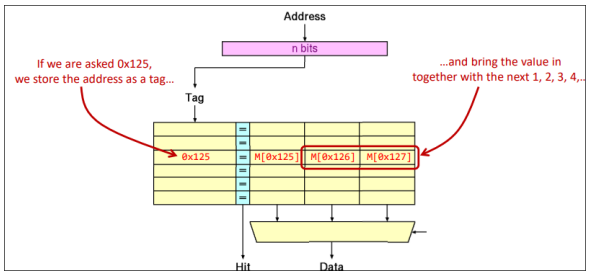
Only exploiting Temporal locality



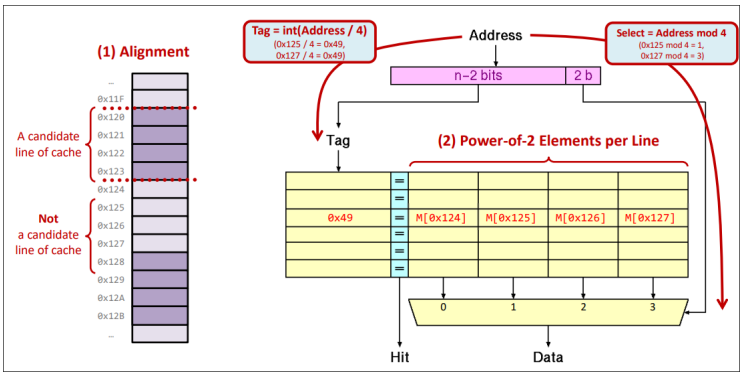
At the current moment we are only exploiting the temporal locality (the upper arrows that goes into data when we are fetching from memory).
A solution here is that when we are fetching the element at address `0x100` I also store the element at address `0x100`, `0x101`, `0x102`, `0x103`.



Why don't we store the address of the first one as a tag? We where storing here for instance `0x40` but wht don't we store `0x100` directly. This works:



However here there is a big issue: if the address we are asking is for instance `0x127` we don't know just from a comparator if the data contains it. What we would have to do is to actually take the range from `0x125` to `0x127`. Which we don't have, it would be too slow. How can we recover from this?
What we can do is to store as a tag only the $n-2$ bits as a tag:

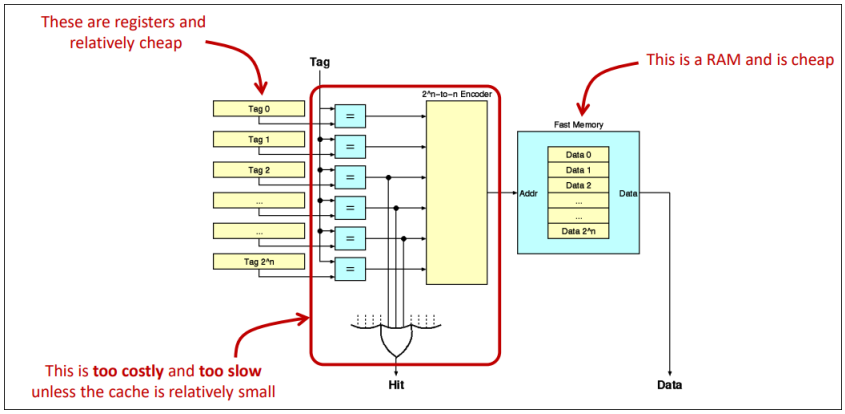


Here all our problems are gone! We only have to do a comparison and the data is easily accessed **and** it is **hardware friendly**. This means that this can be *easily* implementable.

Remark 21. We have a constraint here is that the family of cache has to be adjacent, so that we don't miss any element. Here we will choose all the number divided by a power of 2.
So that by a division by two we can find the family of each element.

Fully-Associative Cache

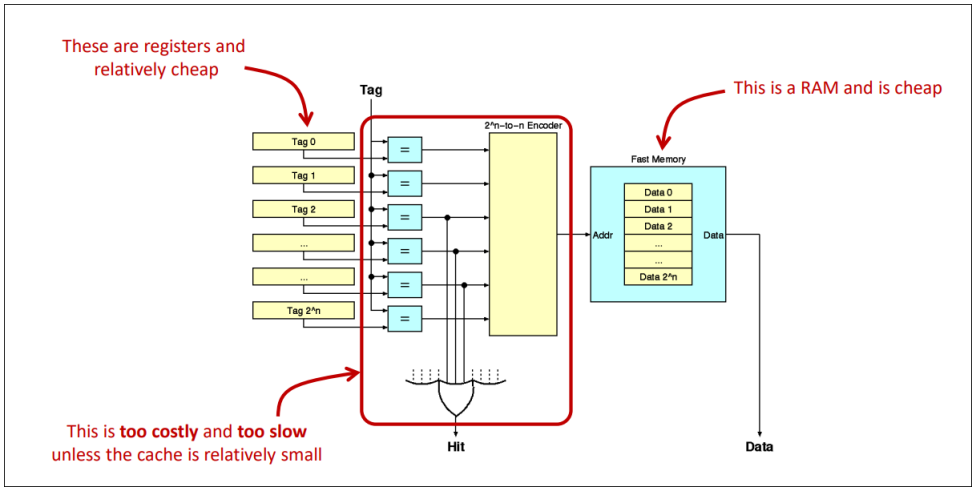
Okay all of this is pretty nice however, there is an issue that has still never been mention, how can we tell to the cache controller when there is a `hit` or a `miss`? The easiest way to do so is to put a big `OR` gate from all the tag. This is **slow**. Imagine having an `OR` gate with a million of input. The time growth of this would be logarithmic which is not good for us. Furthermore, it would be too costly.



Remark 22. When we say here cheap here, cheap is compared to the rest of this circuit which are not cheap at all (`SRAM` is still expensive compared to `DRAM`).

The issue here is that this doesn't work on mega byte of information so we **need** to change something:

How can we make it simpler



Instead of doing what is done one the left, we can do a simpler comparison.
On the right we can do a cheap `RAM` (compared to `CAM`) as the direct output of the `Tag` (have an address of the cache directly as input).

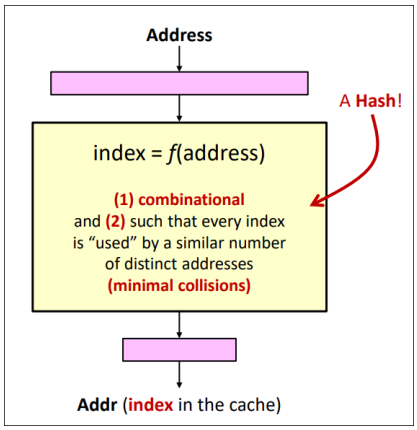
For this to works, every element should go into a **single place** of the `RAM`. I take the address from the processor, I figured the index in the cache, we'll get out **one** candidate word, we have the Tag here, we'll read and compare the tag and the content, if it is a `HIT` then we won else we `MISS`.

Tag

What I found hard to understand here is that how can we know when accessing the memory that we hit? we have an address that in a memory give the output the tag and the content and with the tag we can know if we have a `HIT` or `MISS`?

How to generate `Addr` and `Tag`

Now we are looking at the purple box above:

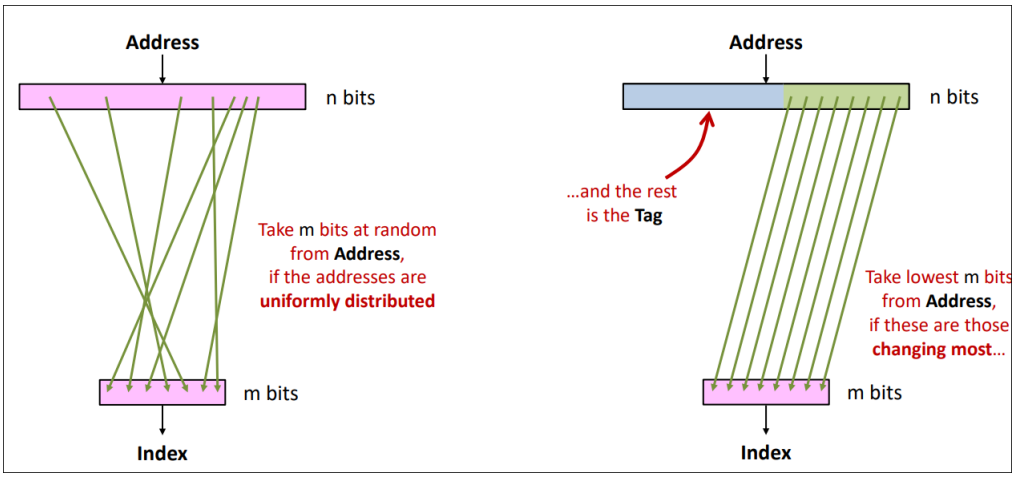


What we want to do is to go from the address in memory which is a 32 bits memory for instance, into a 10 bits memory which is the cache size. How can we do so? We are **Hashing**. We are taking less bits but produce the most different combination possible.

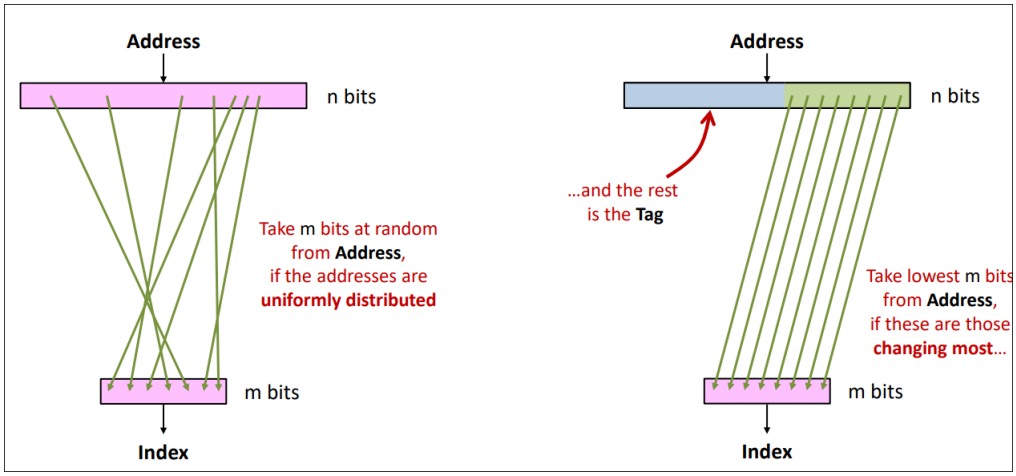
The simplest hashes

What we want as a hash here is a hash that is the simplest and the **fastest**.

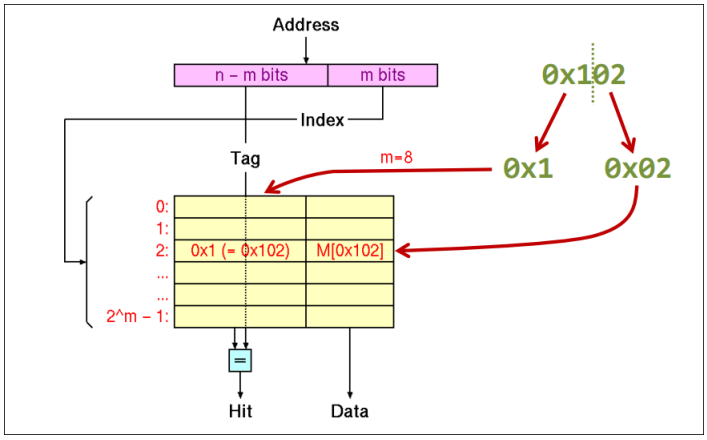
For instance, we can just take some of the bits and we put them in a random (or not) order:



Is this a reasonable hash? It is, **only** if it is uniformly distributed. But when we are writting code or writting data, is the data uniformly distributed?
No: we are always starting at the bottom of the data, we almost never write something at the address `0x3000` millions something (at the end of the memory). We need to take in our decisions the fact that the most significant bits are very rarely used. This is the reason why taking the **least significant bits** is (as we think) the best way of doing it.



Direct-mapped Cache



So here: the index of our address is the lsb bits, then as a tag we put the most significant bit. If we are taking the m bits for the index, then we can take the $n - m$ most significant bits as a `Tag` for our elements, then when we are accessing something we can check if the tag at the index is the same as the $n - m$ msb of our address.

What about it? This is a **very** simple cache here **and** it is **fast** For each access, we only need to do **one** comparison which is very fast.
But is it better that the previous cache:
NO: we have a **lot of constraint**. Here, if we have tow important elements to store in the cache but are in the same hashing index then we are screwed...

Which one is the Best Cache

The question we will try to answer is:

Which one is the best cache?

- Fully-Associative Cache?
- Direct-Mapped Cache?

Example

Consider a **fully-associative** and **direct-mapped** cache, both with 64 **lines** with **four words per lines** $\rightarrow$ 256 words per cache.
The question is how good are they, the criteria for this will be the number of hits.

Remark 23. For this example I found it pretty hard to explain it without drawing so this is the video CS-200-3a. memory hierarchy (30 octoboer 2024 at 46 min).

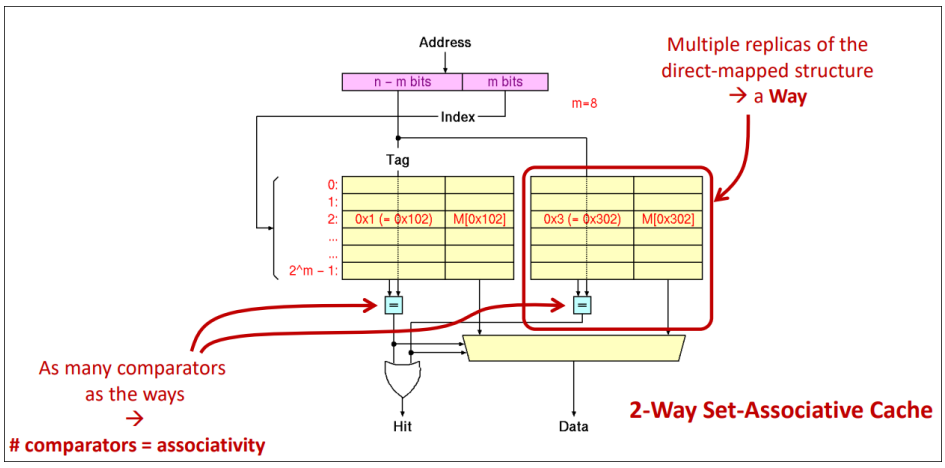
But what we can see here is that sometime the direct mapped cache can be very very inefficient. Is there a way to find the best out of the two worls:
The speed of the direct mapped and the hit rate of the fully-associative one?

Intermediate?

Is there an **intermediate** between:

- **Fully-Associative**: every word can go in every line of the cache (hence the 'full') → associtivity is the number of lines in the cache
- **Direct-Mapped**: every word is 'mapped' to a single line of the cache → associtivity is 1

Set-Associative Cache

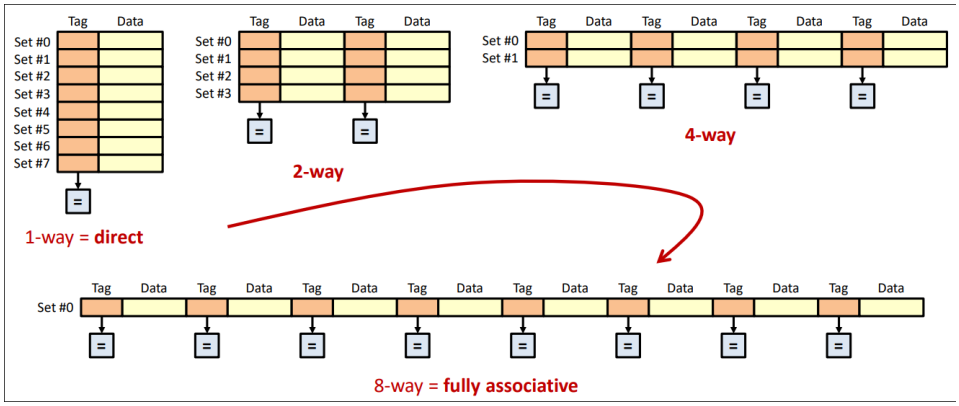


So the idea is to add a second cache map with the exact same direct-mapped structure. By doing this we allow a **second** element for the same index

Critical path Here the critical path is in the **multiplexer**, furthermore every comparisaiton that is made.

Continuum of Possibilities

What we can see here is that we have a **lot** of choice for a given number of storage. For instance if we have 8 storage units we have the following ways of creating a cache:

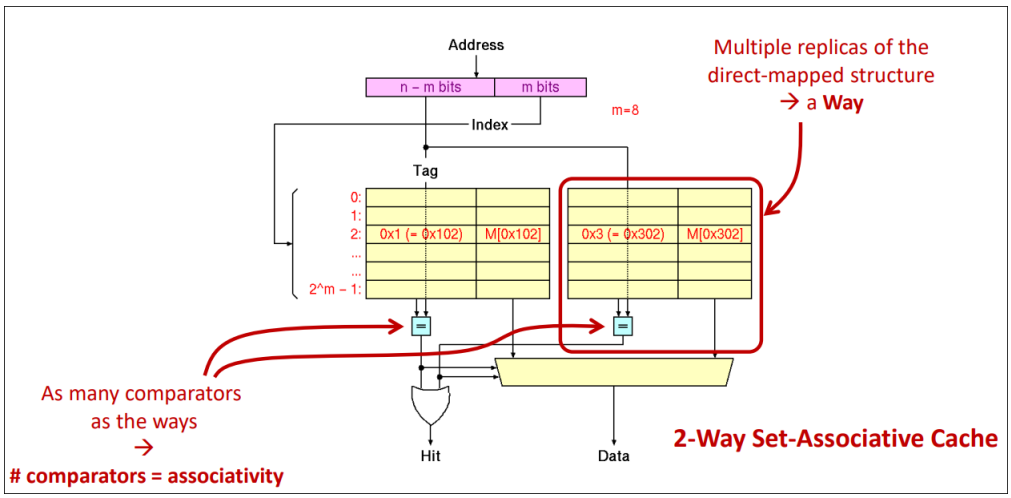


professor's question during the lecture Do we have to have a power of two as the number of way?
Do we have to have a power of two as the number of set?

Remark 24. So here there is something very intresting to see is:
For the first way which is the classical direct-mapped, we only have one comparator, however imagine having to put all the sram next to each other in one set; then we get a cache with 8 comparator $\implies$ We get a fully-associative cache.
We can see it as the direct-mapped is the *transpose* of the fully-associative and vis versa.

Validity

All of this is looks pretty good however we have some issue with the implementation, we always said *If there is nothing there then we put something, if there is something then we get and check*, however: how can we know if there is something in the first place?
In the memory the initial content is **garbage** we don't know if the value that is in the memory is a good one or not.
To fix this issue, all caches need a special bit (**valid bit**) in each cache line to indicate whether something meaningful is in the specific cache line ('0' at reset).



Addressing by Byte

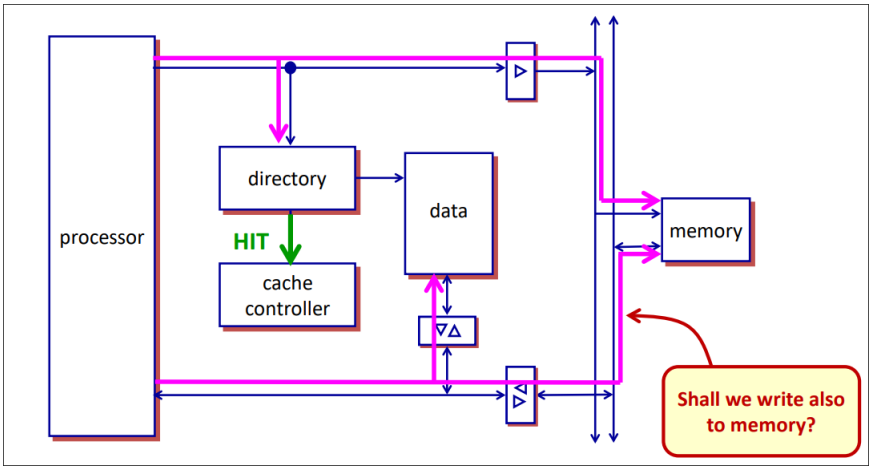
If addressing is **by byte** and **word size is 2^n bytes**, the **n least significant bits** of the address represent the byte offset and are thus **irrelevant**. The last two bits of the address are **internal** to the processor and do not even get to the memory system. As we have seen in Section 1.3.1 the actual way of loading a byte is just a multiplexer between the 4 bytes of a word based on the last two bits of the address. So the **allocation of bits** begin **only** after the second bits. (we do not care about the last 2 bits here).

3.1.1 Write and Cache

Write Hit

If we want to write something in the memory and we have a hit (in the cache controller), the first thing we should do is update the cache so that the next time we are getting something from this address we access the correct value.
The question we have now is: Shall we write also to memory?

This is a legitimate question, by not writting we gain a lot of time. Also if we can avoid to write to memory we let the bus data and address **free**.
However there is a big issue, as we have said before, the cache has only **copies** of what is in memory so overwritting something was not a big deal because the content was still in the memory. Now this is different, we now wrote in the cache **but not** in the memory, this means that the next time we are overwritting in the cache, our data is gone!



Write policies

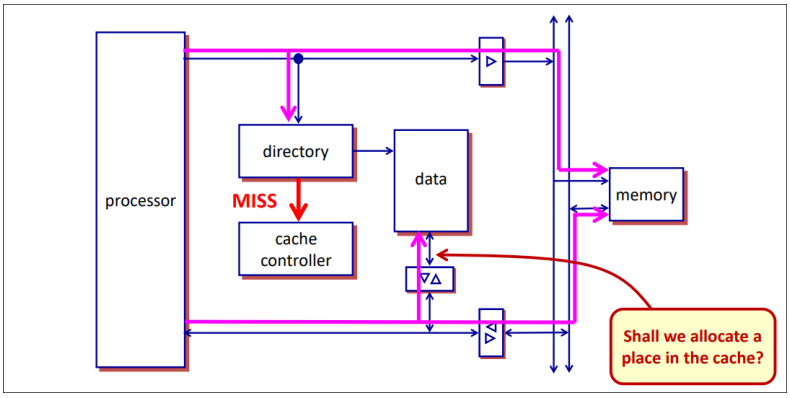
There is a couple of way to fix this issue:

- **Write-through**: On a write, data are always immediately written into main memory
  - (+) Simpler policy
  - (-) May keep the memory/buses busy for nothing
- **Write back** or **Copy back**: on a write, data are only updated in the cache (hence, main memory data will become wrong/obsolete)
  - (-) Needs a **dirty Bit** to remember that cache **data are incoherent with memory**
  - (-) When a dirty line is **evicted**, first it must be **copied back** to main memory
  - (+) we do not have to write evry time into memory

Remark 25. What we are doing here is kind of putting a *flag* that says content has changed and while we don't overwrite in the cache, we don't change the content in memory. This is same principle as lazy evaluation in scala (kind of), we don't change anything until we have to.

Write miss

The question now is: When the cache doesn't have the value yet, shall we put the value in the cache or not?
This question has less *issues* that comes with it, however this is still a legitimate questions:



Allocation Policies

- **Write-allocate**: on a write miss, data are also placed in the cache
  - Simple and straightforward
  - Need to **fetch the block of data** from memory **first**
  - If the processor writes a lot of data that it will never read back, it may **unecessarily pollute the cache**
- **Write around** or **Write no allocate**: on a write miss, data are only written to memory
  - If the processor will load from the same address, it will be a **read miss**

The '3Cs' of caches miss

There is three types of **cache miss**

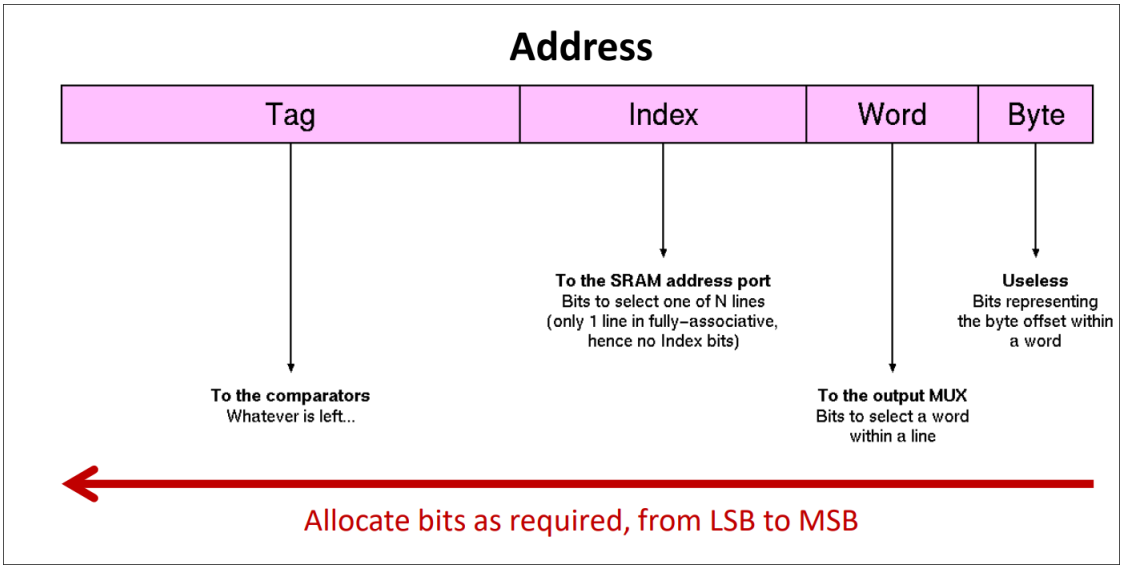
1. **Compulsory** → Missed that would also happen in an infinitely large fully associative cache with the same block (also called **cold-start** misses or **first-reference** misses)
2. **Capacity** → Additional misses that occur because the corresponding block has been evicted due to the **limited capacity of the real cache**
3. **Conflict** → Further misses that occur because the corresponding block has been evicted due to the **limited associativity of the cache**

Those types of caches miss are useful to understand the **source of the limitation of performance**.

Summary of cache Features

Here's the features of a cache and what's make a cache:

- **Cache size**: total data storage (usually excluding tags, valid bits, dirty bits, etc.)
- **Addressing** by byte or word
- **Line** or **block size**: bytes or words per line
- **Associativity**: fully-associative, k-way set-associative, direct-mapped
- **Replacement policy** (except for direct mapped): LRU, FIFO, random, et.
- **Write policy**: write-through or write back
- **Allocation policy**: write-allocate or write-around



3.2 3b: Simple Cache Examples

Compulsory Misses of a Direct Mapped Cache

- Given an initially empty **direct-mapped cache** with **4 lines** and **2 words per line**, find the total number of compulsory misses for the following memory access sequence (in decimal)

$$12, 15, 11, 1, 17, 3, 17, 11, 17$$

- The memory is word addressed
- The least recently used (LRU) replacement policy used

Hits in a 2-Way Set-Associative

- Given an initially empty **2-Way set associative cache** with **2 sets** and **4 words per block**, find the total number of hits for the following memory access sequence (in decimal):

$$11, 15, 4, 16, 3, 10, 12, 24, 19, 25$$

- The memory is word-addressed

- The least recently used (LRU) replacement policy is used

Miss Rate in a 2-Way Set-Associative

- Given an initially empty **2-way set associative cache** with **2 sets**, find the miss rate for the following **block address** access sequence (in decimal):

$$6, 0, 0, 5, 5, 6, 1, 2, 2, 0$$

- The memory is word-addressed
- The least replacement used (LRU) replacement policy is used

Direct-Mapped cache

Consider now a **direct-mapped cache** with a **capacity of 128KiB** and **2 words per line**

- The memory is word-addressed (one word = 4 bytes)
- The memory address has 32 bits
- Show how the address bits are used in the cache

Tag of a 2-Way Set-Associative

Finally, consider a **2 way set-associative cache** with a capacity of 128 KiB and **6<S-**

Remark 26. Those are examples

3.3 3c Virtual memory

Segmentation Fault? Bus Error?

```
#include <stdlib.h>
#include <stdio.h>
int main() {
        int *p = (int *) 1234; \*li t0, 12345*\
        printf("\%i", p);       \*la a0, format*\
                                \*lw a1, 0(t0)*\
                                \*jal printf*\
}
```

This program tries to read at the address 1234? But here is the output:

```
Segmentation fault (core dumped)
```

Here the system sends us a message that this instruction cannot be done, we cannot *steal* an element from memory.
But now at the moment we could steal this data (with our current CPU), for us if we gave an address, the processor gives us the element that is stored at the address. **Overview**

There is **three problems**:

- How to **protect memory** so that each program (processes) running *simultaneously* in the system can only access its own data? How can we isolate processes?
- What happens if the main **memory** (`DRAM`) is **not sufficient** for the execution of program? Can we use our disk? How?
- How do we run several programs (processes) "simultaneously"?
  How do we load **multiple programs in memory**? Where?

This is where we need **multiprogrammed system**

Needs of Multiprogrammed System

- Relocation
  - All programs must be written without knowledge of where they will be in memory
- Protection
  - Programs can access only their own data
- Space
  - If several program run at the same time, memory shortage will be even more a problem

What this means is that for instance gcc compiles a c file, it puts it into the address `0x0` (by convention). However how can we compile more than just one program here (the second would overwrite the first file)? The simple solution is to relocate our load into memory each time

Simples solution: Relocation at Load Time

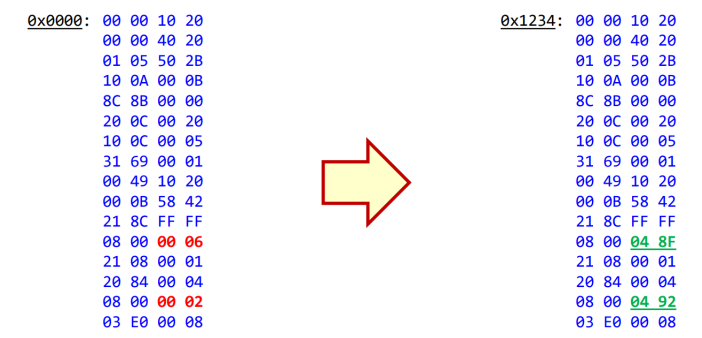
```
0x0000: add v0, zero, zero # v0 = 0
                add t0, zero, zero # t0 = 0
0x0008: sltu t2, t0, a1 # t2 = (t0 < a1)
                beq t2, zero, 0x003C # if (!t2) goto fin
                lw t3, 0(a0) # t3 = mem[a0]
                addi t4, zero, 32 # t4 = 32
0x0014: beq t4, zero, 0x0030 # if (!t4) goto next
                andi t1, t3, 1 # t1 = t3 & 1
                add v0, v0, t1 # v0 = v0 + t1
                srl t3, t3, 1 # t3 = t3 >> 1
                subi t4, t4, 1 # t4 = t4 - 1
                j 0x0014 # goto inner
0x0030: addi t0, t0, 1 # t0 = t0 + 1
                addi a0, a0, 4 # a0 = a0 + 4
                j 0x0008 # goto outer
0x003c: ret # return to caller
```

So if we want to add a new program at this address we can just relocate this one (the code above) at address for instance 0x1234, which will gives us

```
0x1234: add v0, zero, zero # v0 = 0
                add t0, zero, zero # t0 = 0
0x123c: sltu t2, t0, a1 # t2 = (t0 < a1)
                beq t2, zero, 0x1270 # if (!t2) goto fin
                lw t3, 0(a0) # t3 = mem[a0]
                addi t4, zero, 32 # t4 = 32
0x1248: beq t4, zero, 0x1264 # if (!t4) goto next
                andi t1, t3, 1 # t1 = t3 & 1
                add v0, v0, t1 # v0 = v0 + t1
```

```
                srl t3, t3, 1 # t3 = t3 >> 1
                subi t4, t4, 1 # t4 = t4 - 1
                j 0x1248 # goto inner
0x1264: addi t0, t0, 1 # t0 = t0 + 1
                addi a0, a0, 4 # a0 = a0 + 4
                j 0x123c # goto outer
0x1270: ret # return to caller
```

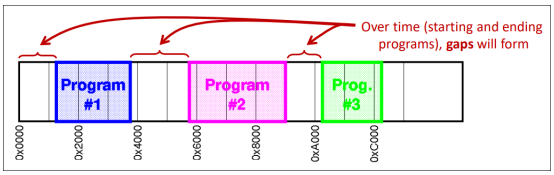
The relocation **must** take place at a **binary** level, not **assembly code**.
We need **relocation tables** to know **where** are the addresses to change



This idea has been used for many many years, and it is still used to this day. This is a simple but very useful solution. However this is still a not the best for us but why?

Limitations

When we are running for instance programm 1, 2 and 3 at the same time we get this in our memory



As we can see there is **gaps** here.
We have a lot of **garbage** that we need to collect to have again some space. For instance if we wanted to add a fourth program and it is bigger than the individual gaps, how can we do it? The solution is the one that we said before, we need a **garbage collector** (which we don't have on our processor nor on our OS).
The limitations of garbage collector is that :

- Large amount of work to do at load time
- **inflexible** → cannot be changed later

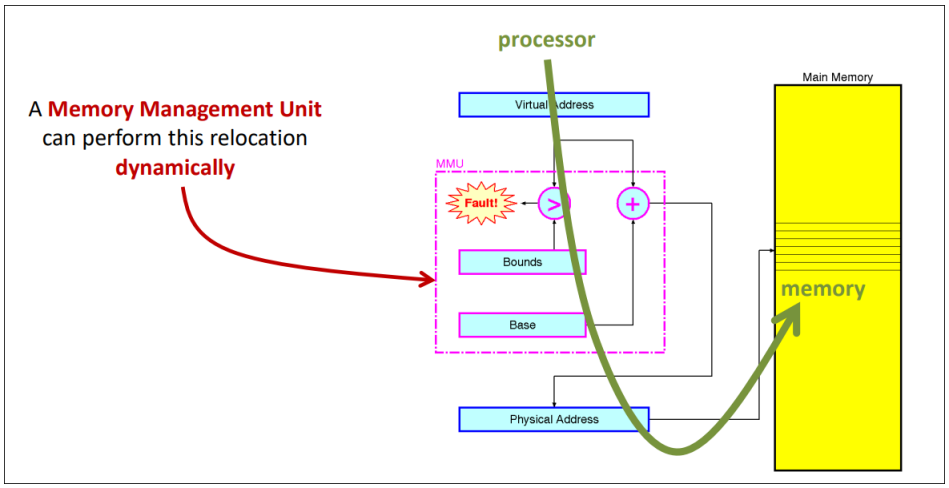
We cannot move any program here. We would think that moving a program is just moving it into a new address and renew the `PC` to the current line that is being executed. **BUT**, what if we are in a function that has a return address which is stored **in the stack** how would we know? From an hardware perspective the memory is **only storing somes bytes here and there**. It doesn't know if it is a program, data, or anything else. Therefore it is impossible to put a program somewhere else **during its execution** and before (I am not 100% for this one please dm me if it is wrong). (Try this with a a program that calls a function into another function (If we changed

everything line by a certain offset then the return that was in the stack is not changed which make the program break.))

Remark 27. What is important (from what I see) is the fact that this is not an issue of something too hard nor too slow. This is just **impossible** to do. we cannot shift a program
The issue is **static**, the fact that we have choosen the address of the program **staticly** makes it impossible to move before, this is the same issue as in Section 1.2.1

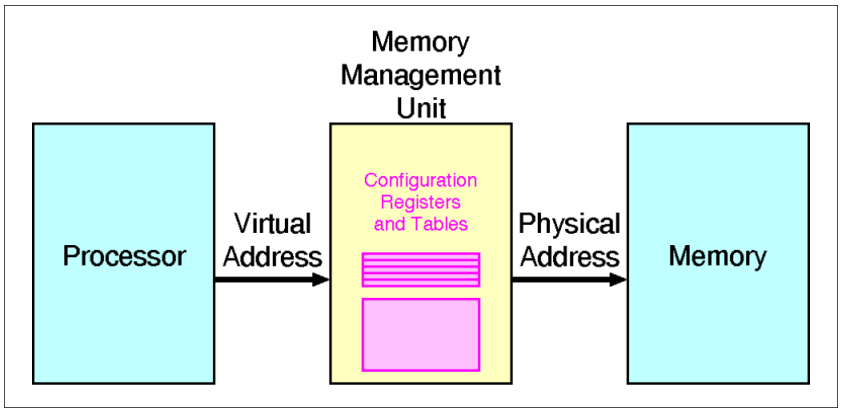
Relocation in hardware:
Base and Bounds MMU

Now, instead of doing it **before** it run, we can do it **while it runs**. What we want is that: from a software perspective we use a virtual address (which thinks that the program start a `0x0`) which is **true** from a software point of view. On the other hand, the program is in fact, at address `0x1234`.



Memory, Management Unit

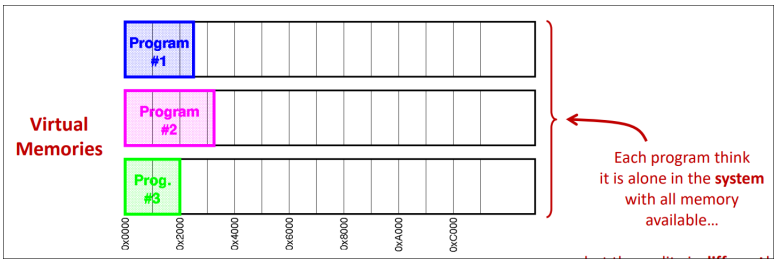
What we have to do is just to put some gates between the processor and the memory in order for the memory to lie to us. We transform **at runtime** the real memory into virtual memory.



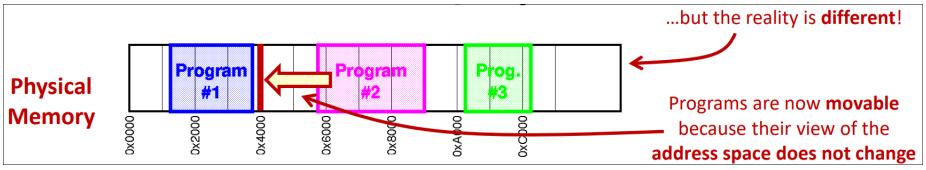
Virtual Memory	This is a memory that the OS allows a program to believe it has.
Virtual address	Conventional address used by a program → the MMU must translate it into physical address at the time of an access
Physical Memory	Memory actually available in the computer
Physical Address	Real location in physical memory; identifies actual storage.

Virtual and Physical Memory

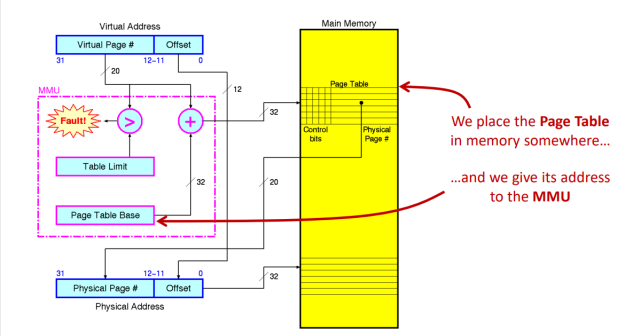
It works now and it is very clean.



But this is only from a software perspective. But in reality (physical memory) this is the exact same thing as in section 3.3. If we are able to do the change on the fly then programs **become movable**. If we return to `0x100`, in respect of where I am in the program, I will always return to `0x100`. Now, the view of the address space does not change.



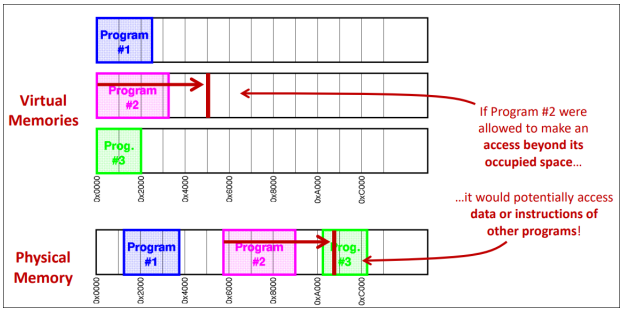
Remark 28. If we take the example of the stack of before, the issue was that when we were accessing returning but the memory was shift then the program was lost. However now, even when we are changing anything in the physical memory, the stack from a software perspective stays the exact same. The MMU handles the translation to wherever the physical stack actually lives at any given moment.



There is two things here: Fist, the offset that is being computed here. Second, a not very costly part which **checks wether or not the address that we are asking** is **too big** or not. The part with the Bounds and Fault is checking if the program is asking for an address that is too big.

Why we check

The reason of this check is very important. For instance if we are the second program and we want to fish something from another program then we can just access something that is further from our scope.



The **Base** and **Bounds** values are different for each programs
On a **context switch** (changing the program running), the OS must **reload these registers**

Needs of multiprogrammed

Now:

- All programs must be written without knowledge of where they will be in memory
  - Space allocation may need **garbge collection**, **moving programs** and **data**, etc.
- Programs can access only their own data
  - Protection is a bit crude: **one chunk of memory**

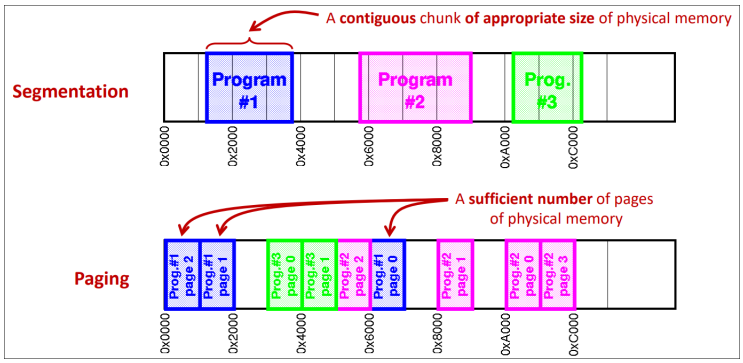
Was is still not good If several programs run at the same time, memory shortage will be even more a problem

Segmentation and Paging

Segmentation An **Segmentation** (an extension of **Base & Bounds**) splits the physical memory exactly as needed by each program

- Artbitrary start of a block
- Artbitrary length
- Multiple block per application

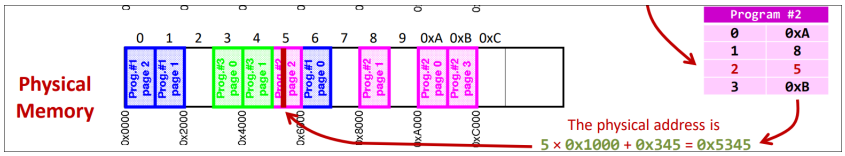
Paging **Paging** splits the memory in equal small block (e.g., 4-64KiB) and assigns as many as needed to each program.



From there, there is a lot of questions on in which page are we now? Where in the current page are we now?
In other word

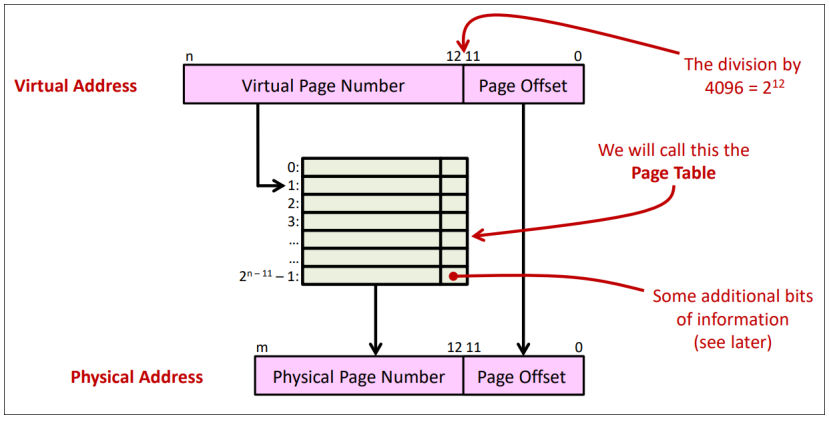
How do we translate?

For instance let us take the previous example and assume that we are in the virtual address `0x2345` and that pages are `0x1000` in size.
To know in which page we currently are we can just do a integer division: `0x2345 / 0x1000` = page 2 and to know the position in the page we only have to take the modulo: `0x2345 mod 0x1000` = `0x345`.
Then we need a **table** that tells us the physical memory of each page.



What if the page's size is a power of 2?

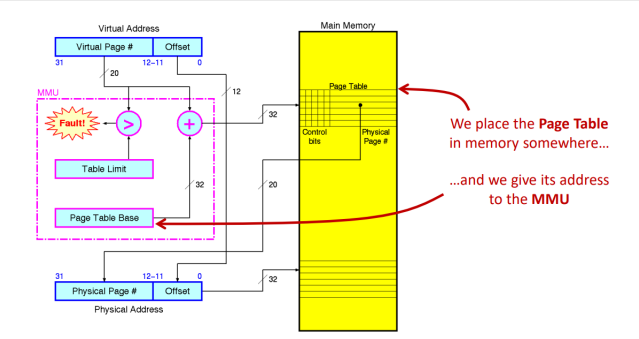
If the length of the page is a power of 2 then this has became totally **trivial**: All we have to do is to take the first n bits for the offset and the $32 - n$ last bits as the input for the page table.



Remark 29. We need a page table **for each programs**.

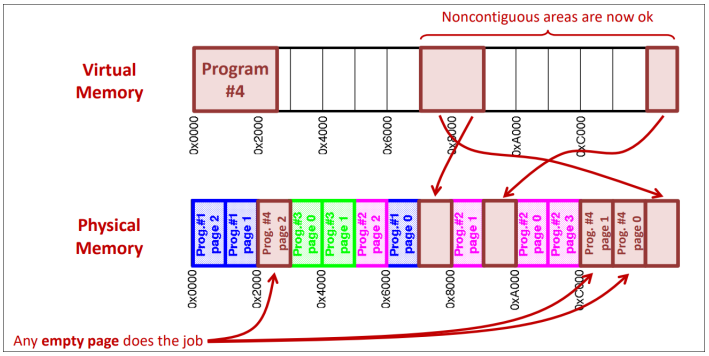
Address Translation in a Paged MMU

The question now is where do we store all those table, In the MMU or in the memory? The answer is the **memory** Instead of having the offset in the MMU we have the address of the page table.



Memory Allocation is easy now

Now what is nice is the fact that there is gap is not an issue now
From a programmer, the address will always be the same (e.g., starting for `ox0`)
Which is not an issue because of the virtual memory. And the the fact of what we put where is not a complicated thing to do, is we have empty space in memory then we just have empty page which can be allocated to a program.



Page Tables Can Be Big

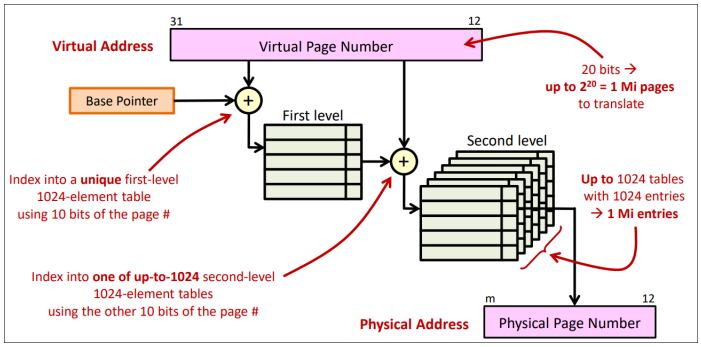
Page table could be very large, for instance 64GiB of memory of memory in 4KiB pages requires 2^{24} entries or approximately **64 MiB**.
For a program that uses only a few MB, **most entries are empty**.
There is a big sparness for the table, those table are gigantic but we most of what we'll use is only a thousand.
As computer scientist we can encode those to optimize. Several possible solutions exist:

- Hashed Tables
- Pages Segmentation
- **Multilevel Page Tables**

Multilevel (Or hierachical) page tables

What we want is a multilevel page stable. Instead of taking the 21 msb bits and put them in a table, we instead take the 10 msb bits as the index of the first table. This first **contains only index** of the second tables. We have one thousand tables that

points to one thousand tables, this means that we have 1 million tables at the end of the day.

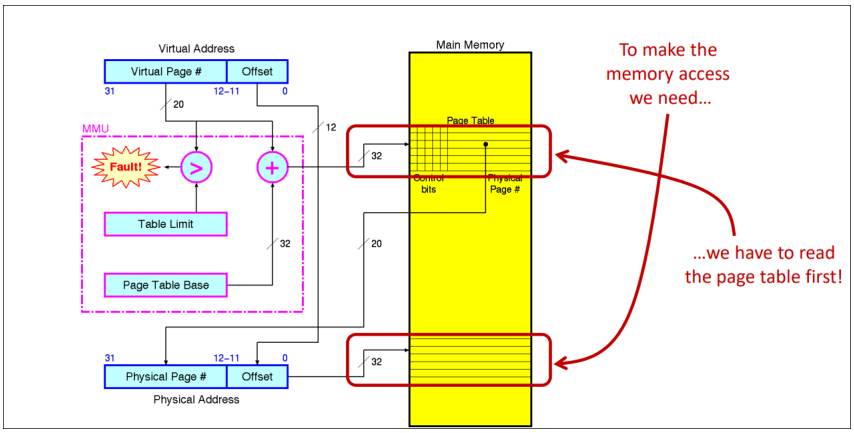


This is kind of weird at first, we are adding a new table, how can we gain time or space by addings something in memory?
But now we don't have to allocate all this space directly in the first place, what we can is only allocate in page tables in memory when it is needed (when the OS decide to allocate).

Remark 30. So here we gain a **lot** of space. The reason why we can do this is that the table is a **sparse table** and we can exploit this in order to use less memory.

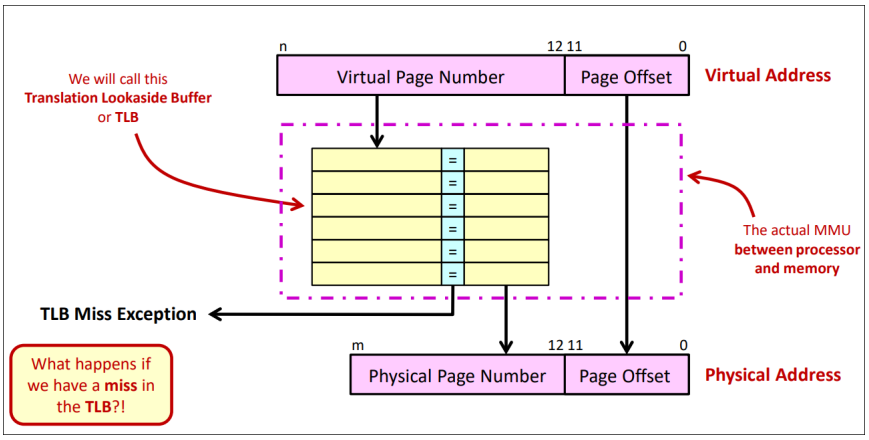
But, Two memory accesses every time?

So we need to have two memory acceses for each access in memory, this is **slow**.



A Specialized 'cache' for the translations

A solution for this is that maybe we can use a cache, but a very specific cache.
How do we do it?



What we want is to do a comparaison in parallel.
Remark, this is `CAM` as we have seen before.
The question now is what should the 'cache' do when we miss? This is an engineering question, we will assume that we usually don't miss. When we say rare here, it means every thousand, millions instruction we would miss.
But we don't really want to implement a cache controller which is something really hard to do (as said before). Instead when we would miss, we'll raise an exception `TLB Miss Exception` this miss that we stopped what we are currently doing, then we access again in the memory the page tables (this can be done with the two table as seen before) we put our element in the cache **and finally** return to the program that was running.

To be a cache or not a cache

This is a cache by the definition we gave before section **??**. But we don't have all the technology with the cache controller etc... This is more a **software management cache**. It depends on the software system to reload some stuff.

TLB Miss

The processor gets an **exception**

- The user's program **stops execution**
- The OS is invoked and **searches** the translation in the **page table**
- If it **does not find the translation**, the user is trying to access memory that has not been allocated to → **kill the program** and we are done
- Otherwise, it **places the translation in the TLB**
- **Restart execution** from the user program's memory instruction that generated the TLB miss
- By construction, this time the **TLB will hit and the user program will continue**

Memory Protection

Typically **Page table** entries have several arritbutes (OS specific):

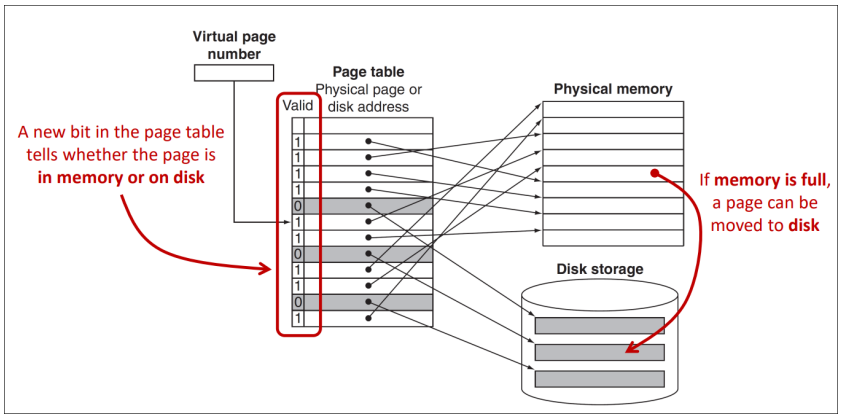
- Valid (to indicate presence in main memory)
- Allocated (to indicae existence)
- Dirty (to indicate a copy-back is needed)
- Used (to help determine which page to replace)
- **Readable**

- **Writable**
- **Executable**
- ...

If the **TLB can be written only by the OS** (e.g, kernel mode), the OS can **protect the Pages Tables** (prevent users from writing them), **protect its code**, and thus control completely **memory access rights**.
So now we have the Relocation and the protection issue **achieved**! The only issue that is left is **space**. The amount of memory that we used before is now splited by a lot of other programs. Memory shortage become even more an issue here.

Not All Pages need to be in main memory

So what can't we just put our data in a disk storage when the memory is full? from the processor this is not possible (the load instruction takes only load from the RAM memory). But now there is the MMU between the memory and the processor, we can make the processor believe that everything is in memory and as the MMU we redirige the address into a load from a disk storage.



So now space is not an issue anymore, the only issue now is performance. We now have practily infinite memory.

TLB miss - Revised

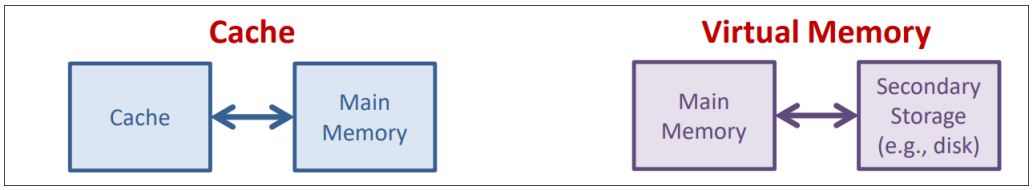
But here we have to rechange our TLB miss because the address can now be also on the disk. this misses that:
When the OS **Reaches the page table** after a TLB miss, now there is a new possibility: the **addressed page is on disk**

- Copy another **page from memory to disk** to make space (Swap, Evict)
- **Bring back into memory** the addressed page (Swap)
- **update** the page table
- **update** the TLB
- Continus as usual

However where are these page on disk? It depends on the OS

- Linux puts them in a special **raw** partition called swap
- Windows puts them in the file `pagefile.sys`

Cache vs. Virtual Memory



Cache

- Cache holds the most frequently used blocks
- If cache is full → evict (LRU, etc.)
- Cache miss → penalty 10x-100x
- A cache block is typically 64 bytes

Some caches can be write-through

Remark 31. A page fault is resolved in **HARDWARE**

Virtual Memory

- Main memory holds the most frequently use pages
- If main memory is fulle → evict/swap (LRU, etc.)
- Page fault → **penalty 100,1000x - 10,000,000x**
- A page is typically 4,096 bytes

Remark 32. A page fault is resolved in **SOFTWARE**

Virtual memory can only be copy-back → dirty bit. We cannot go back and write back into memory this would be a performance disaster. But here what we can do is some clever replacement policies, furthermore timestamps can be implemented (which is not the cae for a usual cache). We can have a lot of help to make our decisions.

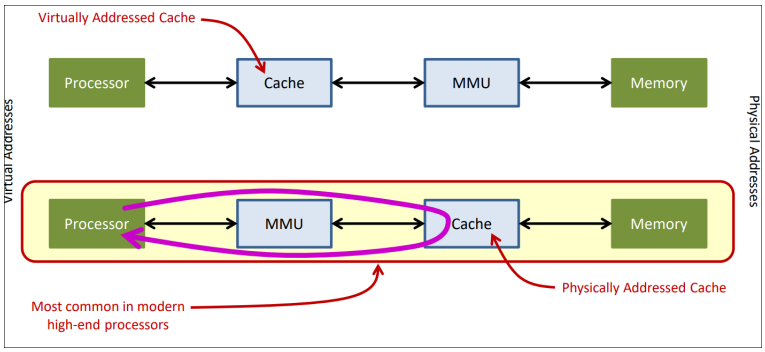
Page table Attributes - Revisited

Typically **page table** entries have several attributes (OS specifig):

- **Valid** (to indicate in main memory)
- **Allocated** (to indicate existence)
- **Dirty** (to indicate a copy-back is needed)
- **used** (to help determine which page to replace)
- Readable
- Writable
- Executable
- ...

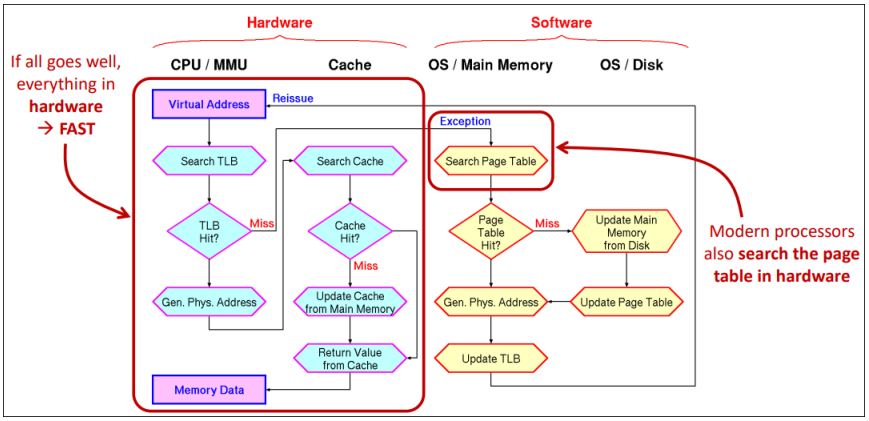
Virtual Memory $\Longleftrightarrow$ Cache

But where do we put fist, the cache or the MMU

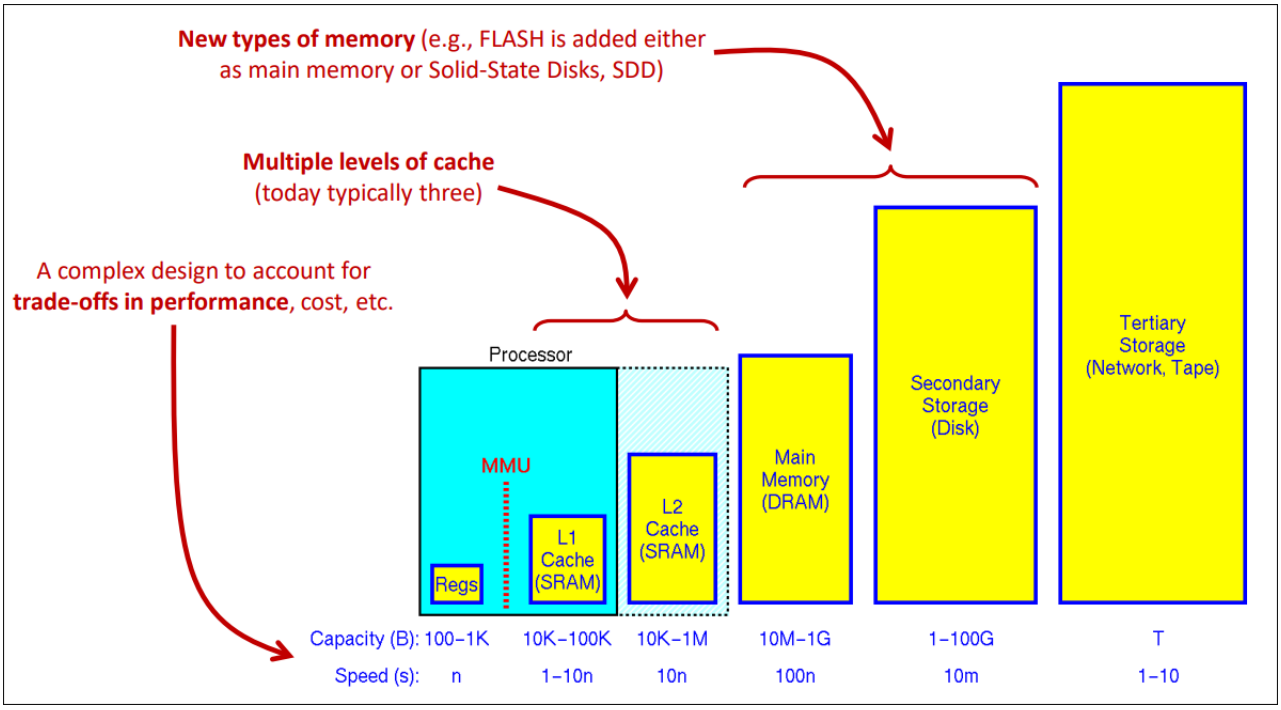


The reason why the most common one is the second is implementation. the MMU translate virtual address into physical address which is then process by the cache. Therefore the cache is the same for every programs while on the other hand if we go the other way around. The cache **is** different for each program. This implies a lot of issue, should we all clear the cache when we are switching program, this looks pretty costly how does the cache know wether or not it is changing of program etc.

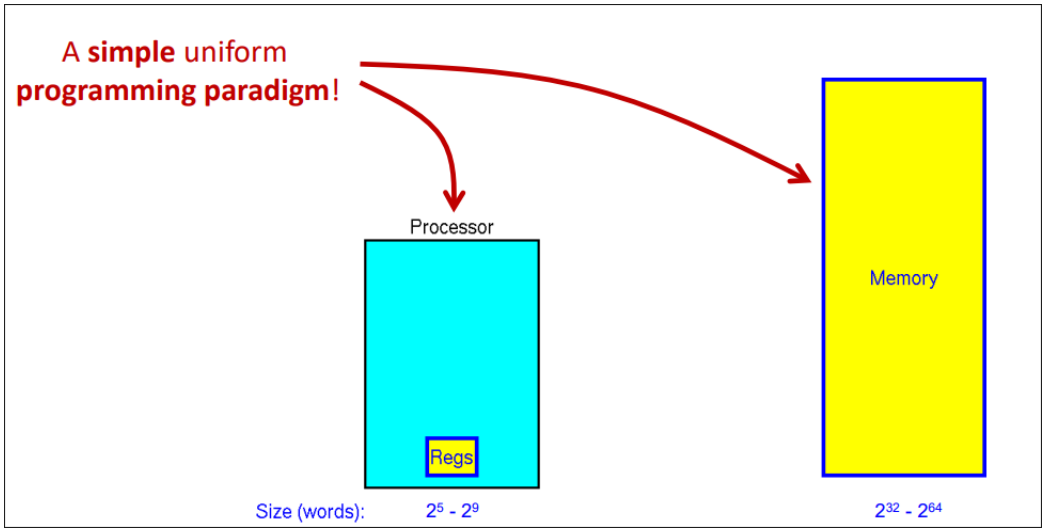
TLB Misses, Caches misses and Page faults



Overall Picture: The System Side



OVerall Piccture: The Programmer Side



3.4 Summary

- **Virtual memory** offers the illusion of a perfectly uniform and identical memory to each individual program
- Additionally, **virtual memory** is a form of caching between main memory and seconday storage
- A **Memory Management Unit** implements mechanisms to translate virtual addresses into physical ones
- **Translation Lookaside Buffers** are special the 'caches' (software managed!) used to perform the translation efficiently in the MMUs
- As with caches, all this is **transparent to users**: programs read and write memory oblivious of all this - and exceptions are used to correct problems
- It is a complex interaction of hardware (MMU, TLB, caches) and software; **exceptions are an essential ingredient**

Remark This part is just me who did my practice here I didn't check the answer nor the typos.. If you want to correct it either send me a dm on telegram or ping on issue on the github LectureNotes

3.5 3d. Simples virtual Memory example

Simple translation Scheme

Consider a **byte-addressable** virtual memory systemm that uses linear page table with **8-KiB pages**, **240bit virtual addresses**, **18-bit physical addresses** and **3 control bits** per page table entry.
One word is **4 bytes**

What is the wifth of the physical page number field in bits?

Answer First let us compute the size of the offset, We have 8-KiB In each pages this implies that the address is contained on : $2^3 \cdot 2^{10} = 2^{13} \implies$ 13 bits. Because virtual memory is only on 24 bits then the remaining bits are $24 - 13 = 11$ However physical address is only on 18 bits which means that we will have to shorten it to: $18 - 13 = 5$ bits

Page Table Entry Size

Consider a **word-addressable** virtual meomry system that uses linear page tables with **16-KiB pages**, **64-bit virtual addresses**, **48-bit physical addresses**, and **2 control bits** per page table entry.
Page table entires are **byte-aligned**

What is the corresponding minimum size of each page table entry, in bytes

Answer We have 16 KiB pages this means that there are 4096 words per pages. This implies that we need 12 bits. For the physical page number we have then: $48 - 12 = 36$ bits and we have 2 control bits $\implies$ 38 bits. Since Page table entry are byte aligned it has to be a multiple of 8 which implies that we need 40 bits or 5 bytes.

Total Page Table Size

Consider a **word-addressable** virtual memory system that uses linear page tables with **4-KiB pages**, **24-bit virtual addresses**, **24-bit physical addresses**, and **1 control bit** per page table entry. Page table entries are **byte-aligned**

Assuming that the page table contains all possible translation, what is going to be its total size?

Answer

We have 1024 words per pages. For the physical address we have $24 - 10 = 14$ bits left with 1 control bit $\implies$ 15 bits. Therefore we will need 2 bytes for the page table entry. For the Virtual page number we have $24 - 10 = 14$ bits which implied that the total number of virtual page is $2^{14} = 16384$. Therefore We know that each table entry has 2 bytes and that we have 16384 of them, this implies that we have

$$16384 \cdot 2 = 32768 \text{ bytes} = 32768 \,\text{KiB}$$

Total Addressable physical Memory

Consider a **word-addressable** virtual memory system that uses linear page tables with **4-KiB pages**. The **physical page number** is encoded on **19 bits** Onw word is **2 bytes**

What is the total size in words of the addressable physical memory

Answer

We have 4 KiB per pages this implies that the page table entry is of size 2048 words. The physical page number is encoded on 19 bits this implies that the total number of page is 2^{19} where each of them has 2^{11} this implies that the total number of word is 2^{30} words.

Address Translation

Consider a **byte addressable** virtual memory system that uses linear page tables with **8-KiB pages**, **32-bit virtual addresses**, **32-bit physical addresses**. The table to the right contains the first 16 elements of a **linear page table** The **Valid** bit indicates that the page is allocated and in memory

Index	Physical Page	Valid
0	0x6235	TRUE
1	0x22BB4	FALSE
2	0x2DE8	FALSE
3	0x34120	FALSE
4	0x1BE42	FALSE
5	0x2D5FF	FALSE
6	0xCC56	TRUE
7	0x6C7B	TRUE
8	0x2ABA	TRUE
9	0xFDFB	TRUE
10	0x3990B	FALSE
11	0x1AB4F	TRUE
12	0x8F0D	TRUE
13	0x1ACE	TRUE
14	0x3465B	FALSE
15	0xB586	FALSE

What is the translation of `0x12A60`
And `0x14C48`

Answer

Here we have $3 + 10 = 13$ bits of offset for each pages. we need $32 - 13 = 19$ bits of page index this implies that we need to take the 19 msb of the virtual address `0x9`. This gives us the physical page `0xFDFB` which is allocated. Therefore we finally need to or it with 13 first bits which gives us:

$$0xFDFB1A60$$

We then need to do the same procedure for the second address `0x14C48`.The 19 msb gives us the number 10 which then is just addded:

$$0x3990B0C48$$

Part II

Final

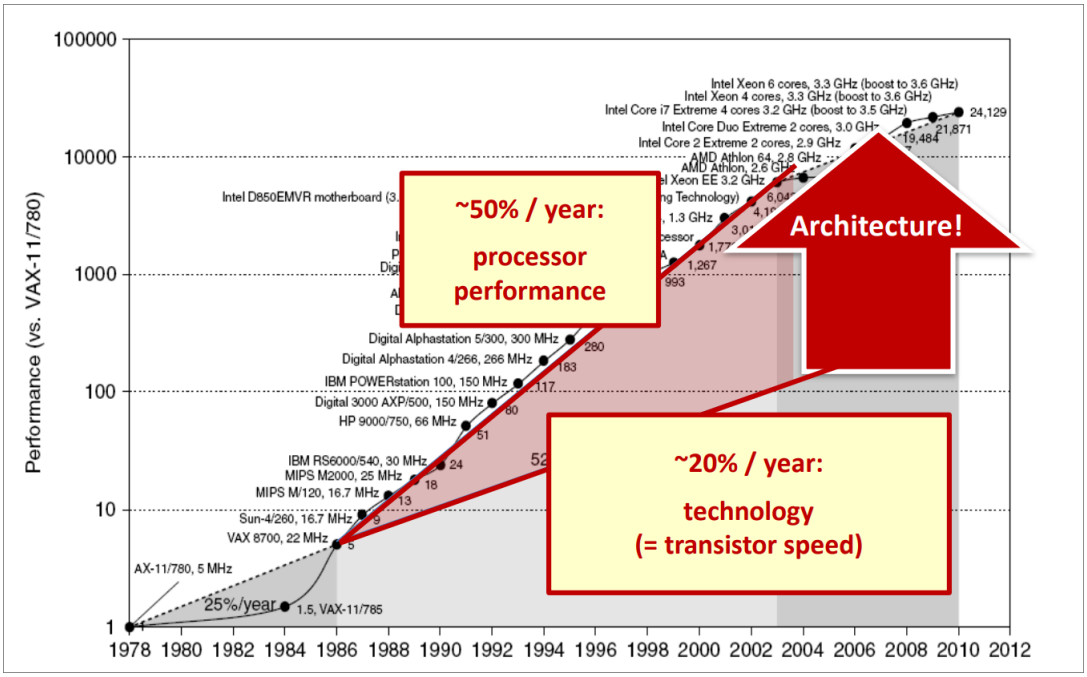
Chapter 4

Instruction Level Parallelism

4.1 Performance

For the past chapters, all we have done is **adding features** to our processor to make sure that it is ready to construct a modern system. We needed some instruction to construct function, jump etc... We also needed some exceptions (i.e. useful to debug our program).
Now we are clear about what we want to do and how to do it.

What should we do now?
We know how to build a processor so let's care about **performance**.



As seen before, we have two ways to improve our performance:

- Better transistors
  - Which is a job that only the physicist, electrical engineer can do.
- More transistors

  - By having better transistors → smaller → cheaper transistors. We can therefore add more transistors for the same prices as before, We can double/triple the number every two/three years.

What and Why?

Now we can have a big number of transistors. The question we will need to answer is: how can we use them in a way to make our processor faster. The answer to that is **parallelism:**
As we have already seen in cs-214, we will use the same principle but at a way lower level

Example: OR

For instance what if we wanted to compute an `or` operation between 20 bits (like an `OR` gate with 20 inputs). A way of doing that would be to `or` each element one by one using only one `or` gate.

- Pros: we only need one gate for all the 20 bits
- Cons: it is **slow**, it runs in linear time

As we said before, the number of transistors is not an issue anymore (kind of).
The better way of doing it is to have a binary tree like of `OR` gates. This way, instead of having to run in a linear time we can run in a **logarithm** one.

Example: ripple carry adder

Another example would be ripple carry adder as we have seen in FDS-173, this circuit costs us the least amount of operation, but is very **sequential**. There is no way for us to compute the last bit without having computed **all the previous bits**. The question we want to ask is: Can we make our ripple carry adder faster by adding some transistors/redundancy?

So Far about Performance Different parts of a system do not benefit equally from manufacturing technology advances:

- Memories are 'slower and slower' → **Caches**

Furthemore at this time in the course, we have done **nothing** to speed-up the processor itself.

What is 'Performance'

But first performance, what is a processor performance?

- **processor frequency**?
  - Is it better an Intel Core i7-770k at **4.2 Ghz** or an AMD Ryzen 5 5600x at **3.7Ghz**? And how much better the best one is?
- Memory speed? Cache efficiency?
  - Is it better to have 8 MiB of 4-way set associative cache or 16MiB of direct mapped cache?
  - Is it better to have three levels of overall smaller chaches or two levels of overall bigger caches?

We need a metric!

Elapsed Time, CPU Time, ..

None of the above matter in itself:
The most important thing for us is how long it takes to perform a job a user needs!
Prof. Ienne is an academic person, so he needs to publish a lot of paper → he compiles a lot of latex file. Let us take the command

```
[110]icvm0100> time latex mypaper >& /dev/null
0.79u 0.17s 0:01.20 80.0%
[111]icvm0100>
```

Remark 33. The time keyword allows us to see the elapsed time form the start until the end of the execution.

- the 0.79u means **User CPU Time**: the processor has spent 0.79s executing instructions of my program (latex)
- **Elapsed Time**: 1.20s after I started it, my job was completed
- **System CPU Time**: The proessor has spent 0.17s executing instructions of the operating system on behalf of my program
- The 80.% here is the pourcentage of the elapsed time that has been spent on my job, the rest has been use for other thing (system I/O, other users, ...)

His job is to write latex paper. he needs to be fast at this job (that's why neovim is the best). Every time he is compiling his work, this means that he has to wait the elapsed time **staring at his screen compiling latex**. This is some time that he is not using for work. So us we are interested in

Elapsed Time on an Unloaded System

Relative Performance

Speedup

For instance we can compute how faster system X is compared to system Y:

$$\text{Speedup} = \frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\text{Execution Time}_y}{\text{Execution Time}_x}$$

Common Performance indices

Here we are talking about benchmarks. How we used to do it is by simulating a program and run this 'fake' program on both system to see which one is better.
This is not very representative on how our system will be doing in the real world right? Why don't we just use a program that **everybody** uses and benchmark with it?
Everybody uses GCC (right?). So the benchmark would look like this:

- We take our machine out of the box
- We install our operating system out of the box
- We take a program written in `C`, `fortran` etc..
- We compile it → we see how much time it takes

The performance of the operating system is also taking in count. We need a operating system that has a very good cache miss handling, etc.

Common indices

- Speedups of systems compared to a single standard system

- SPEC CPU, Geekbench, Cinebench, and LinPack HPL, EEMBC ('Embassy') CoreMark

However with this we have some question when our performance is not good, is it the fault of the processor, the compiler, the operating system?

Personal Remark

The modern computing system is a pretty deep stack:

$$\text{C code} \to \text{compiler} \to \text{assembler} \to \text{machine code} \to \text{CPU} \to \text{caches} \to \text{memory} \to \text{OS}$$

A slowdown could hide anywhere in that stack. When you see that System X is slower that System Y, it might be that:

- the CPU is genuinely slower
- the compiler is not optimized for that architecture
- the OS isn't using the right scheduling policies
- the benchmark's working set doesn't fit in cache on one system
- the filesystem makes tempory files slower
- $\vdots$

A way of catching the cause is to create microbenchmarks that tests only a specific hardware features (cache, ALU throughput). We can also count the cache misses, stalls, IPC, etc.

Remark 34. You can think about all of those benchmark as unit test and integration test as we have seen in cs-214. The integration tests serves to see the overall performance of our processor, OS, etc.

But if that performance is not correct, we will use the small benchmark/ unit tests to detect the bug/bottleneck.

Relate Performance to Hardware Implementation

- In Hardware, our measure of time is the **clock period** or **cycle**
- We are often interested to relate execution time to this 'hardware quantum'
- **Cycles per instruction (CPI)**
  - Average number of cycles per instruction executed

$$CPI = \frac{\left(\frac{\text{Execution Time}}{\text{Clock Period}}\right)}{\text{Total Instruction Count}}$$

- **Instructions per Cycle (IPC) ← 1 / CPI**
  - Average instructions ececuted per cycle
  - Normally below unity, unless the processor executes several instructions in parallel

Improving Performance?

Performance being 1/Execution Time, rewriting the definition of CPI and IPC:

$$\begin{aligned}\text{Performance} &= \frac{1}{\text{Exectution Time}} \\ &= \frac{f_{clock}}{\text{Instruction Count} \cdot CPI} \\ &= \frac{f_{clock} \cdot IPC}{\text{Instruction Count}}\end{aligned}$$

So for us the best thing we could do would be:

- Implement the processor in a fast technology (improve f_{clock})
- Execute several instruction in parallel (improve IPC)

Many other Considerations Influence the Performance

Théorème 1. Amdahl's Law Law of diminishing returns
The performance enhancement possible with a given improvement is limited by the amount the improved feature is used

Typical Software situation If a programs sends 20% of the time in subroutine X, the maximum reduction in execution time one can get from optimising x is 20%, that is a speedup of 1/(1 - 0.2) = 1.25x

In a processor If the instruction y is used 0.1% if the time, is it worth to make it faster? It is probably better to look for the instruction which is used 20% of the time..

Benchmarks

Performance indices such as SPEC CPU, Geekbench, CI • System CPU Time: The proessor has spent 0.17s executing instructions of the operating system on behalf of my program • The 80.spent on my job, the rest has been use for other thing (system I/O, other users, ...) My job for me is to write latex paper. I need to be fast at this job (that's why neovim is the best). Every time I am compiling my work, this means that I have to wait the elapsed time staring at my screen compmiling latex. This is some time that I am not using to work. So us we are intrested in Elapsed Time on an Unloaded System Speedup For instance we can compute how faster system X is compared to system Y: Performancex Execution Timey Speedup = = Performancey Execution Timex Commonformancedices Per- in- Here we are talking about benchmarks, how we used to do it is by simulating an program and run this 'fake' program on both system to see which one is better. But this is not very representative on how our system will be doing in the real world, why don't we just use a program that everybody uses and benchmark with it? For instance GCC, everybody uses gcc. So the benchmark would look like this: • We take our machine out of the box • We install our operating system out of the box • We take a program written in C , fortran etc.. • We compile it → we see how much time it takes This takes also the performance of the operating system in count. We need a operating system that has a very good cache, etc.. Common in- • Speedups of systems compared to a single standard dices system • SPEC CPU, Geekbench, Cinebench, and LinPack HPL, EEMBC ('Embassy') CoreMark However with this we have some question when our performance is not good, is it the fault of the processor, the compiler, the operating system?nebench, and LinPack HPL, EEMBC('Embassy') CoreMark need a **precise definition of the user job(s) to run**. Serious **benchmark suites** are collections of large and representative user programs spanning all areas of typical use, often agreed between manufacturers. They do not only define the programs, (in C, C++, FORTRAN, Java), but also how to compile them, what data to run them on, etc.
So for instance this is what a SPEC CPU2017 benchmark look like (SPEC CPU is the most common one today)

SPECspeed	SPECrate	Language	Application
600.perlbench_s	500.perlbench_r	C	Perl interpreter
602.gcc_s	502.gcc_r	C	Gnu C compiler
605.mcf_s	505.mcf_r	C	Route planning
620.omnetpp_s	520.omnetpp_r	C++	Discrete event simulation—computer N/W
623.xalancbmk_s	523.xalancbmk_r	C++	XML-to-HTML conversion via XSLT
625.x264_s	525.x264_r	C	Video compression
631.deepsjeng_s	531.deepsjeng_r	C++	AI: alpha-beta tree search (Chess)
641.leela_s	541.leela_r	C++	AI: Monte Carlo tree search (Go)
648.exchange2_s	548.exchange2_r	Fortran	AI: recursive solution generator (Sudoku)
657.xz_s*	557.xz_r	C	General data compression
603.bwaves_s	503.bwaves_r	Fortran	Explosion modeling
607.cactuBSSN_s	507.cactuBSSN_r	C++, C, Fortran	Physics: relativity
Not applicable†	508.namd_r	C++	Molecular dynamics
Not applicable†	510.parest_r	C++	Biomedical imaging: optical tomography
Not applicable†	511.povray_r	C++, C	Ray tracing
619.lbm_s	519.lbm_r	C	Fluid dynamics
621.wrf_s	521.wrf_r	Fortran, C	Weather forecasting
Not applicable†	526.blender_r	C++, C	3D rendering and animation
627.cam4_s	527.cam4_r	Fortran, C	Atmosphere modeling
628.pop2_s	Not applicable‡	Fortran, C	Wide-scale ocean modeling (climate level)
638.imagick_s	538.imagick_r	C	Image manipulation
644.nab_s	544.nab_r	C	Molecular dynamics
649.fotonik3d_s	549.fotonik3d_r	Fortran	Computational electromagnetics
654.roms_s	554.roms_r	Fortran	Regional ocean modeling

As we can see here there is a new Spec which is the rate which measure the number o ftasks performed in the unit of time (throughput test)
What w ecan also be intrested in is also how do OS and the compiler etc.. works well with the memory, virtual memory

Benchmark	Language	KLOC	Resident size (Mbytes)	Virtual size (Mbytes)	Description
SPECint2000					
164.gzip	C	7.6	181	200	Compression
175.vpr	C	13.6	50	55.2	FPGA circuit placement and routing
176.gcc	C	193.0	155	158	C programming language compiler
181.mcf	C	1.9	190	192	Combinatorial optimization
186.crafty	C	20.7	2.1	4.2	Game playing: Chess
197.parser	C	10.3	37	62.5	Word processing
252.eon	C++	34.2	0.7	3.3	Computer visualization
253.perlbmk	C	79.2	146	159	Perl programming language
254.gap	C	62.5	193	196	Group theory, interpreter
255.vortex	C	54.3	72	81	Object-oriented database
256.bzip2	C	3.9	185	200	Compression
300.twolf	C	19.2	1.9	4.1	Place and route simulator
SPECfp2000					
168.wupwise	F77	1.8	176	177	Physics: Quantum chromodynamics
171.swim	F77	0.4	191	192	Shallow water modeling
172.mgrid	F77	0.5	56	56.7	Multigrid solver: 3D potential field
173.applu	F77	7.9	181	191	Partial differential equations
177.mesa	C	81.8	9.5	24.7	3D graphics library
178.galgel	F90	14.1	63	155	Computational fluid dynamics
179.art	C	1.2	3.7	5.9	Image recognition/neural networks
183.equake	C	1.2	49	51.1	Seismic wave propagation simulation
187.facerec	F90	2.4	16	18.5	Image processing: Face recognition
188.ammp	C	12.9	26	30	Computational chemistry
189.lucas	F90	2.8	142	143	Number theory/primality testing
191.fma3d	F90	59.8	103	105	Finite-element crash simulation
200.sixtrack	F77	47.1	26	59.8	Nuclear physics accelerator design
301.apsi	F77	6.4	191	192	Meteorology: Pollutant distribution

Summary

- Performance measurement is all but easyL many **heterogeneous parameters** come into the picture
- What really matters most for a user is the **elapsed time on an unloaded system**
- **CPI** and **IPC** help relate hardware features of the processor to performance, but there are **many pitfalls**, hidden dependencies, 'second order' effects..
- **Benchmarks** are the only practical way to assess performance–and serious unbiased benchmarks are difficult to design

4.2 Basic Pipelining

Circuit Timing and Performance

Most of the time so far, we have mainly discussed circuits at a higher, level of timing abstraction: what happens every **cycle**

- Finite State Machines: `state <= next_state`
- Function units and memory elements perform one operation over a small number of cycles; e.g., a combinational ALU performs an addition per cycle

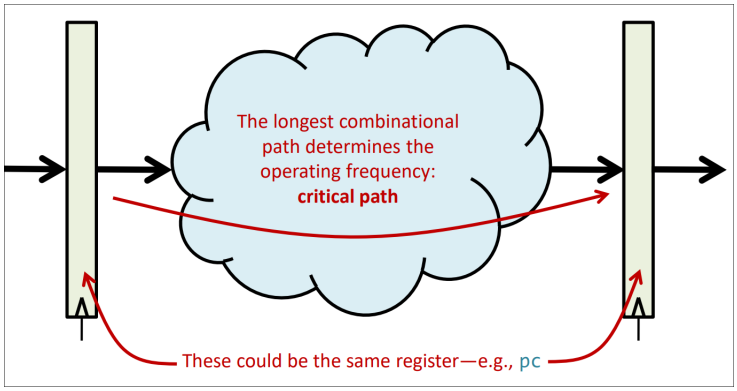
To make faster circuits, we need to zoom-in briefly and understand more of **signal propagation** and timing limitations

Signal propagation

The edge of the `clock` signal indicates:

- When **new data can be applied** to the combinational part of the circuit (from the left register to the cloud)
- When old input data have crossed the combinational part of the circuit, the result is ready, and it **can be stored** at the output (from the cloud to the right register)

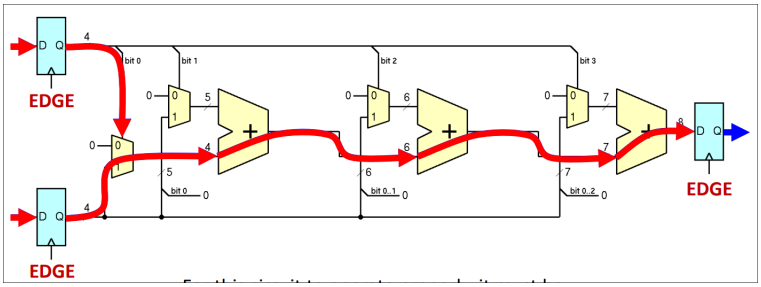
To operate circuits 'as fast as we can', we apply a `clock` signal whose **period is equal to the critical path delay** (that is, the longest delay) of the circuit



The cloud here is the abstraction of any circuit that we are doing.

Example

So if we for example take the following circuit:

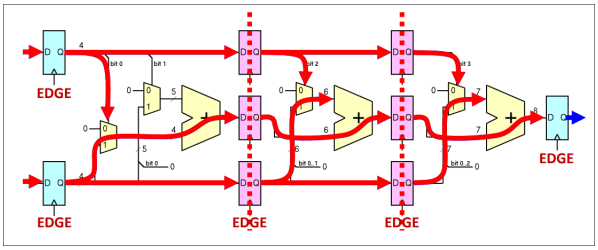


Remark 35. What does this circuits do? let us analyse:
First we will check each bits of the upper register.
If the first bit is one we put it into the first adder. If the second bit is one we also put the the lower register in the adder but shifted of 1 (to the left) which makes is 3·lower register. If we take the third bit (the 2 bit in the circuit) then this time we need to shift the lower register by two instead of one. etc..
At the end of the day this is a multiplier!

For this circuit to operate properly, it must be:

$$T_{CLK,comb} \geq T_{\text{critical path}}$$

A solution in order to make the time of the critical path smaller is to add intermediate registers for instance:



Here we divided our big circuit into three subcircuits which each of them have at most the previous critical divided by three. Therefore, for this new circuit to operate properly, it must be:

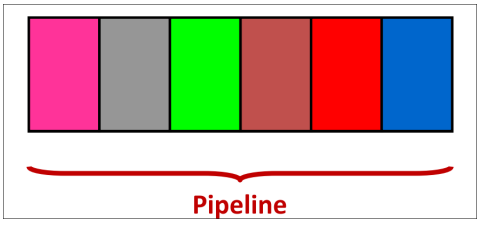
$$T_{CLK,pipe} \geq T_{\text{new critical path}} \simeq \frac{T_{\text{critical path}}}{3}$$

What has changed?

In terms of functionallity nothing has changed, all we did is divided the clock which has no impact on the output of our circuit.
But now our **clock can run faster**. If we have introduced N stages of intermediate registers evenly, roughly our critical path is N times smaller. this directly implies that our **clock can run N time faster**!
But is this great? Not really, yes our clock run N times faster but now we also need N cycle to do the same thing.

So why?

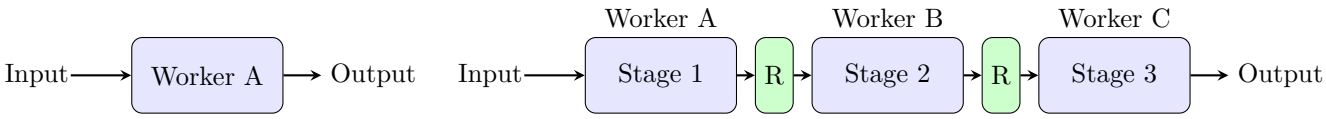
So now we can have multiple Operations in a pipeline, we can use the inactive areas for one operation for **other operation**



Remark 36. This is a very important concept to understand.
By adding registers, we move from an **"artisanal" production model**, where a single worker builds the entire product from start to finish, to a **large-scale assembly line**, where each worker performs only a very small part of the process.
Adding registers is like adding more workers to the assembly line. Instead of one person doing all the work, we now have N people working in parallel, which means the system can deliver results up to N times faster.

Artisanal Approach
One worker does complete task

Pipeline Approach
Task divided into specialized stages



This means that now, every worker need a third of the original period. So every third of the previous period we are able to output something → we go three time faster.

Any Advantage Now?

The time to compute a single operation is **roughly the same** as in the orignal circuit. But now we have new result that are available:

- In the original circuit, **every original period T**
- In the circuit with the registers used for a single calculation, **every N cycles of period $\frac{T}{N}$** → every T
- In the circuit with the registers where we inject a new computation every cycle, we get:

a new result every T/N!

We can generate arbitrarily more result (N large results)?????

Latency and Throughput

Definition 5 (Latency)**.** Time between a computation begins and result is available

- Original circuit: T
- Pipelined circuit: $\frac{T}{N} \cdot N = T$

Definition 6 (Throughput)**.** Number of results available in the unit time

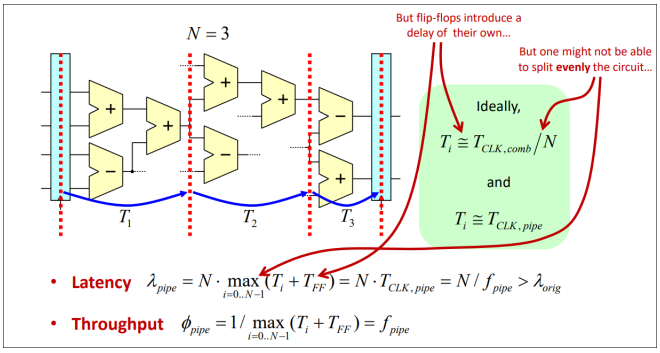
- Original circuit: $\frac{1}{T} = f$

- Pipelined circuit: $\frac{1}{\left(\frac{T}{N}\right)} = \frac{N}{T} = N \cdot f$

Remark 37. All of this looks great but they only look this great in theory, maybe there is some practical issues...

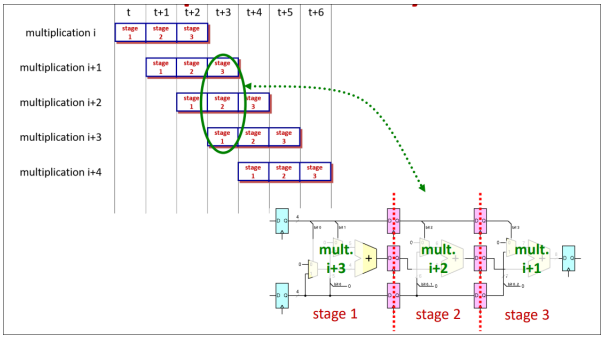
Practical Pipelining

When we start cutting our circuit, imagine we cut it perfectly by three, then the critical path of this circuit is the maximum between this sub three paths.
Now if the critical path of each subcircuit is magically three, then yes this would works but what if we cannot?
Imagine if we cannot cut perfectly our circuit in third equally subcircuits (which is usually the case), instead of having the critical path being the sum of the three critical paths, we would need to put the maximum critical path among the three multiplied by three.
Furthemore by adding a register we alse take the delay of this register into our critical path.



Useful Representaiton of the Pipeline Activity

So here what we see is that in one cycle we actually do three sub operations, which means that every clock cycle, we are able to output something that take us normaly three clock cycles. This is **parallelism**. Not necessarly in the way of how we see it but it still is!



Summary

- **Pipelining** consits in splitting a task in smaller 'subtasks', and in performing in **parallel** each 'subtask' on a different piece of data
- **Pipelining** is an extremly general technique to **increase the throughput** of a system (circuit, processor, computer, ..)
- **Pielining does not improve latency** (actually worsens it!)
- Therefore, pipelining is only effective when one has to reapeat a job on many pieces of data (signal processing, communication packets, and **processor instructions**!)

4.3 Pipelining

As said in the previous section, pipelining comes down to splittting our circuit into subcircuits by adding registers. those registers store the informations of the previous subcircuits which makes it free to compute something else. This is that part of pipelining that makes us win a lot of time, we use the pipeline registers to free the previous part of the circuit. As said before there is a lot of backdown which makes it not perfect (see Summary (4.2))

Pipeline for processor

Pipelining is useful only if the activity in object needs to be repeated many time (as in a production line)

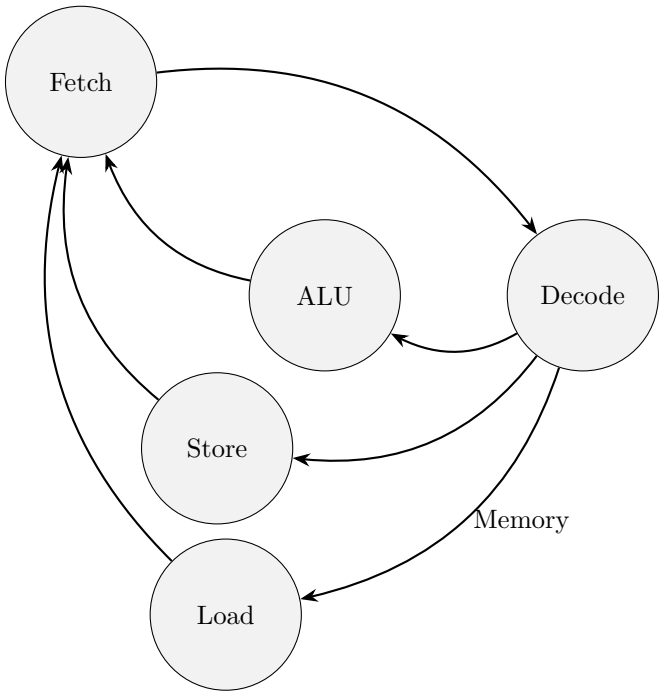
- We have plenty of **instructions** to execute

Pipelining needs to split the single activity in object into many subactivities

- We have a **good logical split** into fetch decode, execute, etc.

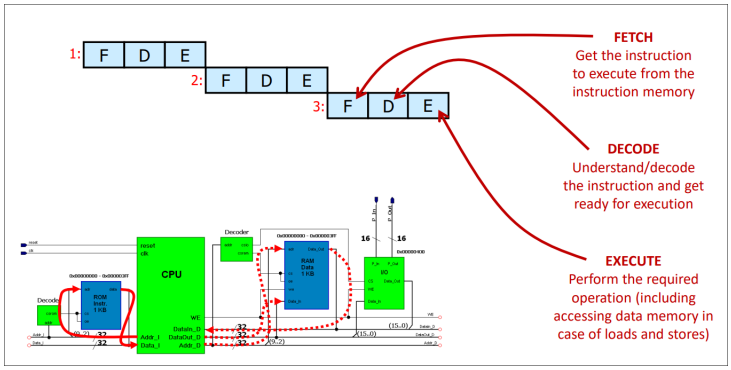
So we have already done some pipelining in lab b, all the state that we added (load1, load2, etc.) is in some way pipelining.

A Simple multicycle CPU



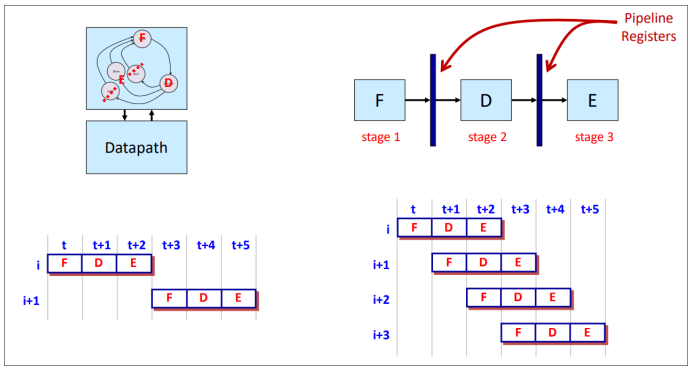
A Simple schedule

At the current moment we are doing pipelining but wihtout any pipeline registers. We are still in sequential mode.



Pipelining the processor

The question we want to ask now is: how far are we from acutally pipelining?
We have already split our finite state machine, what we want now is to actually use those split to gain time:



So right now we are on the left side of the images, we have every instruction that is still done sequentially. What we want is to go to the right side.

Remark 38. Be careful: the left one which is a **finite state machine**, this is not a circuit, this is an abstract representation of the working of the circuit. We will kind of mix them up (the fsm and the circuit) but they are not the same thing.

So here what `F` is storing is all the information of what the cpu needs to do (reading, branching etc.), after `F` is done (fetch instruction), whatever is activated after this one is going to be put in the second block `D`. Which means that after `D` has received the information, `F` is free again of working on its own → compute the fetch of the next instruction

No Hardware In a multicycle processor, some hardware components may be shared across **states**:

- **FETCH** typically requires an **adder** to increment the program counter
- **EXECUTE** naturally needs an **ALU**
- They are **never used at the same time**, so the ALU can be used to increment the program counter

In a pipelined processor there cannot be sharing across **stages**, in general:

- All stages are **active all the time**
- Hardware needs to be **replicated** where appropriate

Two Main Problems

4.3.1 CISC vs. RISC

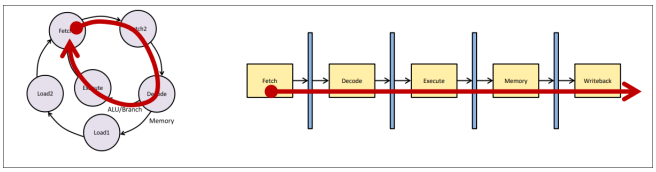
Can we **build equally well pipeline** for a Complex Instruction Set Computer as for a Reduced Instruction Set Computer?
And what does this even means complexe, reduced?

FSM vs. Pipeline

If we look at any loop, at any cycle, it is an instruction, any of the path **represent a different instruction**.

- **FSM**: The number of state I go from is the number of cycle I will use for each instructions.

- **Pipeline**: Whatever of the instruction, the instruction will have to go through all the pipeline, go through all the state.
- **FSM**: **Any path** through the FSM represents the **sequence of necessary steps** for the execution of **an** instruction
- **Pipeline**: **The ordered path** through the pipeline is the **sequence of all possible steps** for the execution of **any** instruction

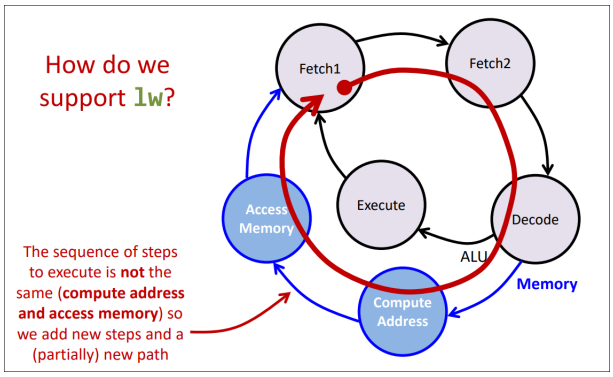


Adding an instruction to a Multi-cycle processor

Imagine that we only have one instruction in our multi-cycle cpu `xor` which result on the red arrow on the above image. What if we need to support the `add` instruction? To do so we wouldn't need to add any more state, the sequence of steps to execute is the same (fetch instruction, read registers, use the ALU, save result in the register), we just need to an **ALU that can perform additions**.

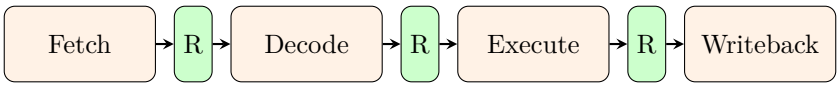
Adding an instruction: not so great instruction to a Multi-cycle processor

Imagine now that we want to add the `lw` instruction, this time we will need a diferent path for this:



Adding Instructions to a Pipeline Processor

So for the `xor` and `add`, the pipeline doesn't need to be changed.



But what if we wanted to add the `lw` instruction again? Then we need to add a new tube into our pipeline:

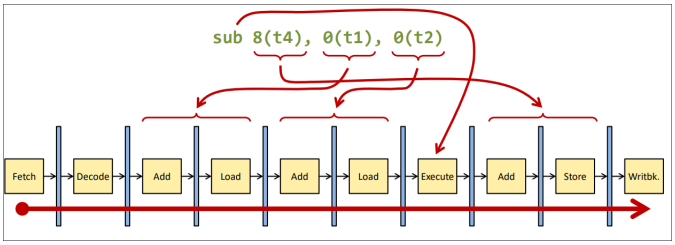
But here this is not really good, imagine that the `lw` instruction was very rarely used, we would have changed our pipeline, just to add an instruction that is used maybe 0.1% of the time. We would have slowed our latency by one cycle for an instruction that is barely used, that doesn't seem very efficient...

The importance if the ISA

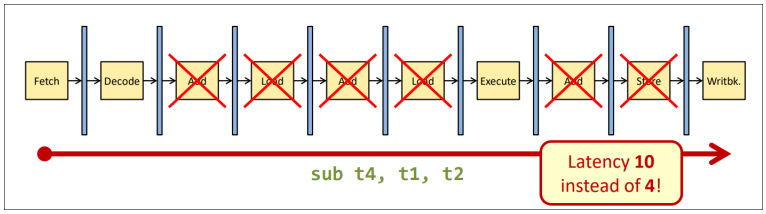
Imagine that we want an to have an instruction (we are abusing the RISC-V syntax)

```
sub 8(t4), 0(t1), 0(t2)
```

This is a very bad instruction for us (as RISC-V programmer) but equivalent instruction exists in x86. This is a cisc instruction.
But now if we go back to see how the pipeline for this instruction would look like we can see that this is horrible:



This is very sad, imagine now that we try to execute the instruction `sub t4, t1, t2` There 6 stages that are just **useless**



This is the reason for us to use **Reduced Instruction-Set Computer** instead of complex one.

Reduced Instruction-Set Computer

Instead of imposing a **huge penalty to every simple instruction** by making complex instruction possible, let's **have only similarly simple instructions** and build our programs with those.
Instead of writing `sub 8(t4), 0(t1), 0(t2)` we would divide this instruction into 4 sub instruction:

```
lw t3, 0(t1)
lw t5, 0(t2)
sub t3, t3, t5
sw t3, 8(t4)
```

It turns out that is is not the only way to go, but it is a **good one** and we will follow it...

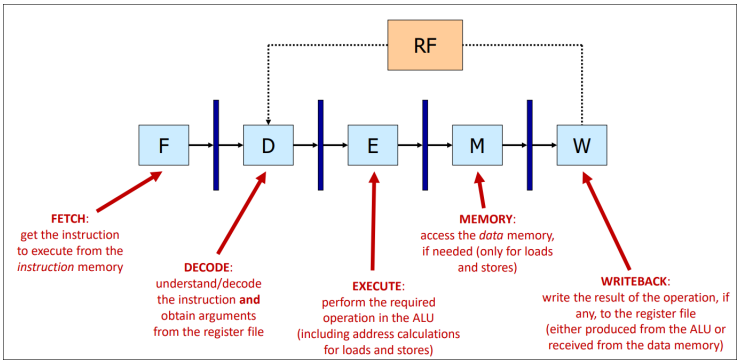
Remark 39. So the only way to do pipelining efficiently is to have simple, **uniform instructions with similar execution paths**. If instructions differ too much in structure or in the number of steps they require (as in CISC designs), the pipeline must include additional stages to support them, and **every instruction—common or rare—pays the cost** of those extra stages. This increases latency, complicates hardware, and wastes cycles.

By contrast, a Reduced Instruction-Set Computer keeps all instructions short, regular, and easy to decompose. This makes it possible to design a pipeline where **each stage does a small**, predictable piece of work, and all instructions flow through it smoothly. Complex operations can still be performed, but they are built from multiple simple instructions that the pipeline already supports efficiently.

In short: RISC enables clean pipelining; **CISC fights against it**. That is why modern pipelined processors—and the teaching architecture we study—choose RISC-style instruction sets

The question we have and that we have already seen before (see 4.3), is: What if instructions are not independent, are we able to execute code **correctly**? **Simple 5-Stage MIPS Pipeline**

As an example, we wil take the first pipelined processor to be commercialized: MIP's



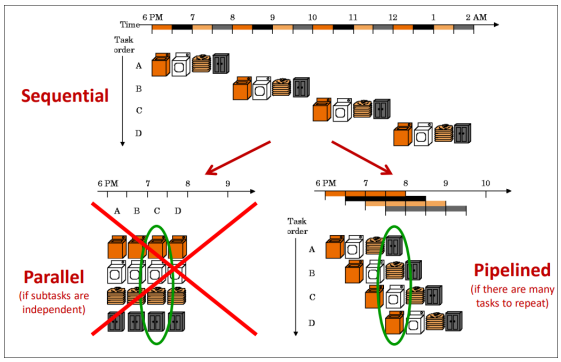
Remark 40. The reason of the `0(t2)` syntax in RISC-V. Whenever accessing or writing to memory, we always go first into `E` which is execute, which means that we actually get the addition for free here. Even if we didn't want to have that addition we would still need to go through the EXECUTE so might as well always add and add 0 when it is not needed.

The Laundry Metaphor

The point here is when we have a job a sequence of things to do that you want to parallelize, some of those things are just impossible to parallelize. However in the case where parallelizing is hard to do, pipelining is very often the solution.

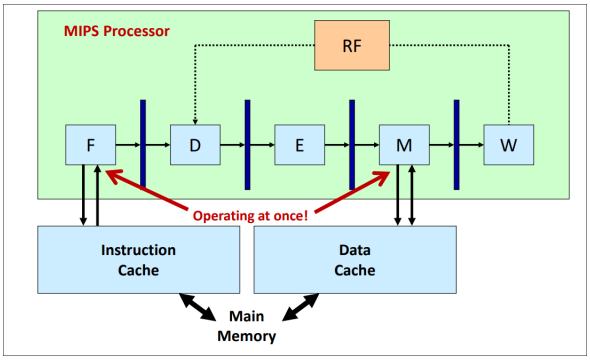
For example, imagine we haven't done laundry for a whole month. We would have to run about four loads. We would love to do four of them in parallel but we only have one drier, one washing machine, etc.

The normal way of doing it is to launch one machine wait for it to finish. When it is done we relaunch a new load and dry the one that is clean, etc. We are pipelining our clothes!



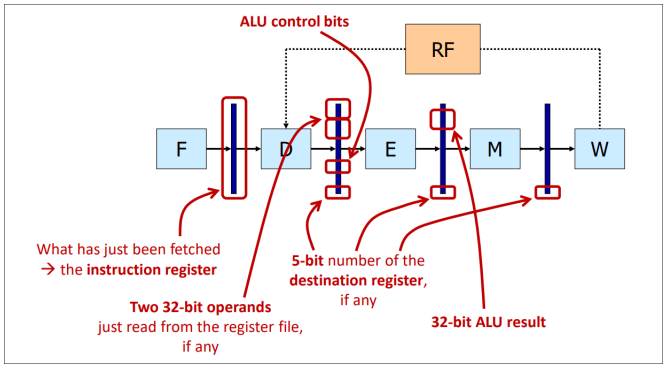
Two distinct memory interfaces

MIPS (and most modern CPUs) usually separate **instruction memory** and **data memory**, at least logically in a pipeline. This is called the **Harvard architecture**.
The reason for this is the following:
At the FETCH stage, we read instruction from memory. In MEMORY stage we also read/write in memory. But remember what we said before (see 4.3) we cannot have two stages sharing the same circuit. This means that we need to separate them → instruction memory and data memory. But we cannot split our memory in two. Instead of doing two **distinct memories** we actually use **two separate caches**.



What is in the Pipeline Registers

By definition the pipeline registers stores **all data that is needed later**. Every bits, ALU, wire, .. That will be used/needed later in the circuit **has to be stored** in the registers.
This means that usually those are the things stored in the registers:



Example of Pipelined Execution

For this part I won't be screenshooting 10 slides to explain the animations but you can find the explanation in the video *cs-200 –4x. Instruction Level Parallelism Paolo Ienne* at 52:30.

4.3.2 Instruction are not independent

All of this is nice I agree but imagine the following code:

```
addi $r0, $r0, 1
sub $r2, $r0, $r1
```

Now try to run this with a pipelined processor... OUCH! In the decode we need a value that is being computed in the EXECUTE, which is far away of being put in the register file.

RAW, WAR and WAW Dependences

Remark W is for 'write', A is for 'after' and R is for 'read'.

Dependency are an important thing that we will cover. Let's look at the code:

```
divd $f0, $f1, $f2
addd $f3, $f0, $f4
subd $f4, $f5, $f6
addi $f0, $f5, 10
```

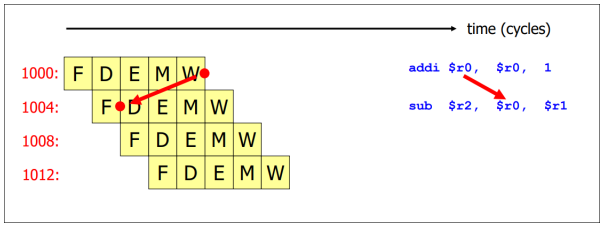
In our code we have that:

- `add` has a **RAW** dependence on `divd`
- `subd` has a **WAR** dependence on `addd`
- `adddi` has a **WAW** dependence on **divd**

The first dependence is a **data** dependency whereas the two other are **name** dependencies.
All of those dependencies gives us the information that those instructions has to be done sequentially, we have to wait for the other instructions to be done before starting a new one. But wait, is there really an issue with the name dependencies here? Not really, there are *kind of fake*. All we have to do to make them disappear is to change the name of the registers.

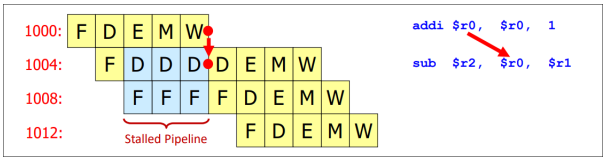
As we can see the first one is way more severe that the others.

Data Hazard



This is a **causality violation**. We try to use a result before it is produced.

Data Hazards Solved by Stallig the Pipeline



The natural solution to **Data Hazards** cause by **RAW** dependences is to implement some logic in the processor to stop/repeat the decoding until the required value is available.

'**Stalling**' roughly means introducing `nop`'s in the pipeline (we are making the code sequential again)

Due to the rigidity of the pipeline, if one stage is stalled (`D` in the example), all the preceding ones must be stalled too (e.g., `F`).

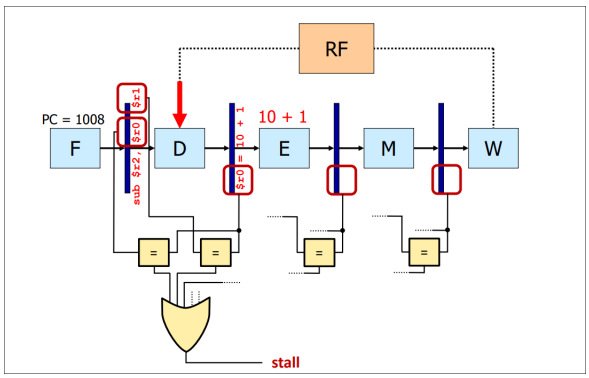
So for us what we need to do is those two things:

- We need to understand that we have a poblem (Detecting)
- We have to resolve the problem by running what is missing (Stalling)

Detecting

The question we will try to answer is how do we see that we have a problem here?

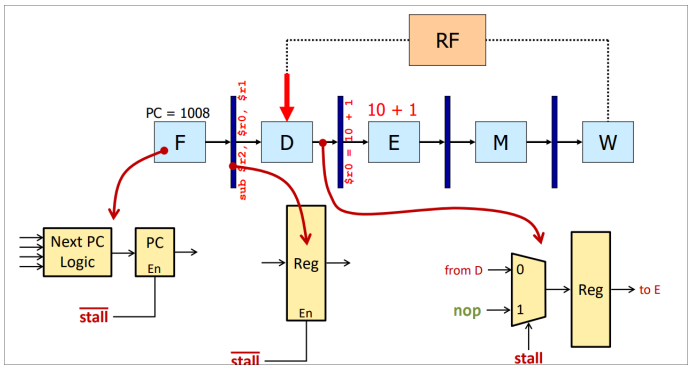
What we need to check is if the output of the EXECUTE is the same as the input at the DECODE. If they are equal then we have a problem. But this is not all, we also have to check for the other pipeline register, if any **pair** is detected then we have a problem.



Stalling

We assume that we know that there is a problem. What we need to do is to let the pipeline compute the current instruction (wihtout fetching a new one) (finish the execute). We would need a multiplexer to before the EXECUTE which would input a nop instruction instead of a rubbish instructions. After that we also need to not enable the pc and to redo the decoding of the instruction with maybe the new value.

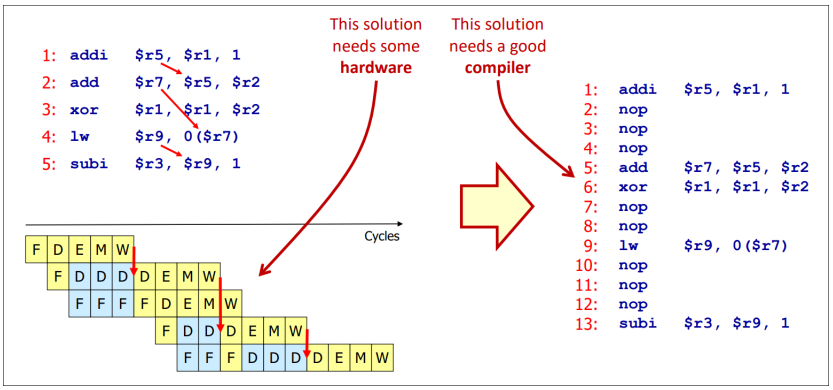
So here we block everything before the DECODE stage. We will block it until nobody tells me that there is an issue.



Now we have an example in the course which is on the course *cs-200 – 4c. Instruction Level Parallelism (cont'd)* on November 2024 at 17:00.

Another Solution

What we were doing is adding `nop` everytime our CPU need to stall from the hardware. But adding those nop is pretty easy to detect before running the program right? so why can't you do it **form the software**?



I means this doesn't look very hard to do in the software right?
But is it better?
Let us take here the line 3 of the left code, the `xor`. It is independent, which means that in the perspective of the stalling, it comes down to the same thing as a `nop` instruction right? So we can replace a `nop` by an independent instructions → we gained one instruction for **free**. This is a very nice compiler's problem! If we are able to build a very good compiler that is able to detect those kind of case, this means that we are able to save a pretty good amount of instructions!

Architecture and Microarchitecture

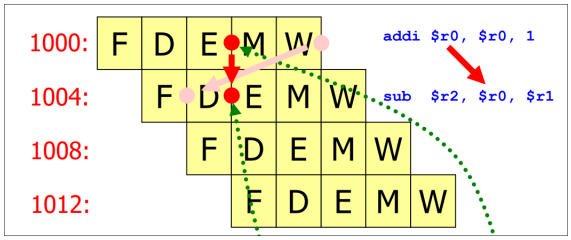
What we are doing is **microarchitecure**, we are trying to opimize our CPU to do less stalls as possible.

- **Architecture**: what is in the ISA contract
  - Instructions, registers, etc.
- **Microarchitecture**: what is specific of an implementation
  - Multi-cycle vs. pipelined, FSM or pipeline structre, etc.

However those solutions we evoked before (reshedule instructions, add `nop`'s, delay slots) **expose typically microarchitectural aspects** (pipeline structure) **in the architecture** → the same binary **does not run** on different processor

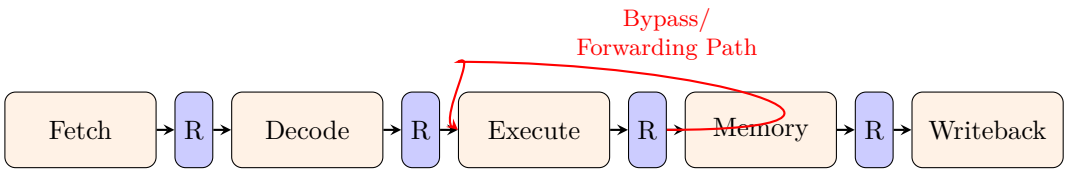
Data Hazard Solved by Forwarding Values

So now our pipeline works but can we make it better? For now at every stalling we loose all our progress.
Let us go from the example



So here we have a `sub` right after an `addi`. If we take our pipeline the result of our add is actually computed right after the EXECUTE stage:

So our issue here is that we need the output of the EXECUTE at the input of the EXECUTE. Might as well create a bypasse path for this right?



So now we need to add a multiplexer for us two know if we need to take the forwarding path or not.
However this is a little glitch here: What does those register contains? Imagine that:

- We are forwarding E → E (ALU result to the next ALU input)
- M → E forwarding (Data memory output or ALU result from M stage)
- W → D forwarding (register file writeback in same cycle as read)

Now the 'glitch' happens in the W → D register file forwarding. The register file must:

- **write** a value in W stage
- **read** it in the D stage of the next instruction

in the same clock cycle

This is just **impossible**.

In order to resolve this issue we have to check in the DECODE if the register that we took from is valid or not → we need some bits of information that tell us wether the register we are using is valid or not.
Us from two pages ago would take those valid bits as redundancy, we need the **value** and **register**.
We then check with a multiplexer if the register value is rubbish or is it valid. If it is rubbish we then use one of the red arrow, else the value.

Classic MIPS pipeline

For a classic 5-stage pipeline with **all forwarding paths**:

- E → E, M → E
- W → D

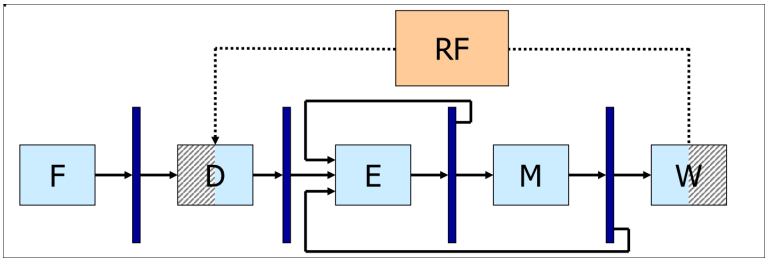
The **register-file forwarding** (W → D) is a special case:

- During **W**, registers are written in the **first half** of the cycle
- During, **D**, registers are read in the **second half** of the cycle.

How can we divide our clock cycle

The way of doing it is use level sensitives, we write in our register-file when there is a rising edge of the clock, we read our register-file when there is a lower edge of the clock. The write occurs when `clk` is high, the read occurs when the `clk` is low.

This means that a register can be correctly written and read in the same cycle



Example again on *cs-200 4c. Instruction Level Parallelism November 20, 2024 at 1:05*
But here we have saved a lot of stalling, our pipeline is almot perfect. The only stalling we still get is when we are decoding a register that is in E but need to be loaded for instance:

```
lw r9, 0(r7)
sub r3, r9, r1
```

So here **most of the stalls** are removed.

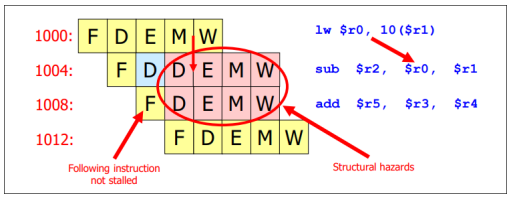
Remark

This is only something that can be done in hardware, this is not possible to do from a software perspective.

Structural Hazards

Definition 7 (structural hazard)**.** A **structural hazard** happens when different instructions compete for the same ressource (e.g., pipeline stage)
It is a **ressource conflict**.

Our structural hazards **canot happen** in our pipeline. If we did not stall also instructions following one missing an operand, we could have structural hazards.



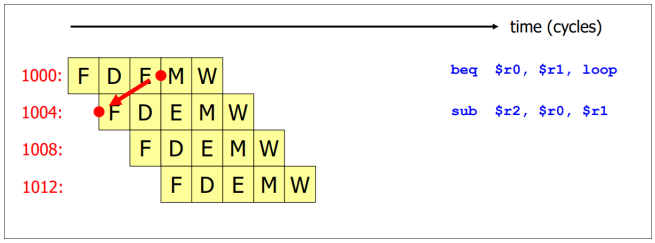
But what happen if we have a **cache miss**? If we have a data cache miss, then we have to wait for the cache to responds, all the previous stage are stalling, the W stage on the other hand can advance (the same principle goes for the F stage).

What about miss on both side?

So here if we have two miss then we have a **structural hazard** on main memory. The way of solving them is to let the most right miss have the priority.

Control Hazard

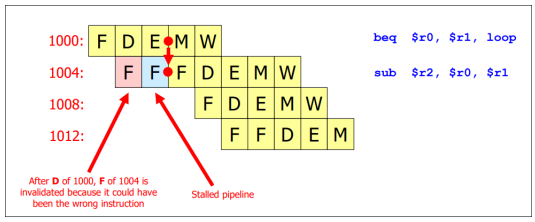
Let us take for instance this code:



Here we have a big issue, after the branch instruction, what should we fetch? We don't know which instruction we should have fetched. This is a **causality violation**

Control hazard solved by stalling

So here the easy way of solving it is to stall the pipeline when we have a branch.

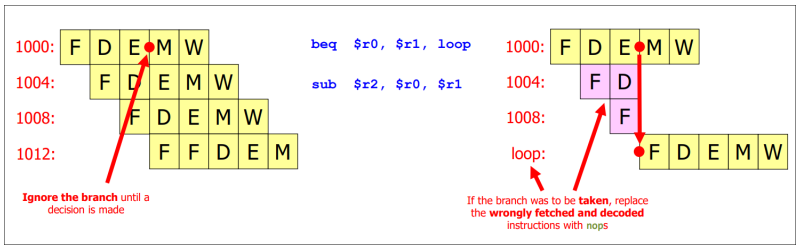


Similarly to the way we solve data hazards, we can **stall the pipeline** (F), one it is discovered, after **D**, that an instruction was a branch, and this **until the branch is resolved**.

If, for instance the correct address of the next instruction is know at the end of the E stage, **2 cycles are lost every branch**

Fetching and decoding do not do any damage

Maybe we can have a slightly better view of that. After all we are not fetching rubbish in the first case. There is a possibility that the instruction that we have fetched is actually correct.

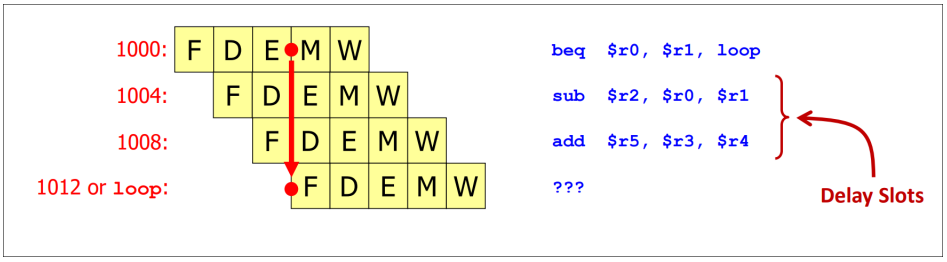


- Fetching or decoding a wrong instruction does not create any problem, provided that the instruction is not also **executed**
- If the outcome of the branch is known at the end of the E stage, we can **wait** until then to **conditionally kill** the following two instructions in the pipeline **if the branch happens to be taken**
- Now **2 cycles are lost only for taken branches** and **none is lost for nontaken ones**

Another solution

As we have seen for data hazard, we can also use the software in order to resolve those hazards. We could just add `nop` at this good place which would resolve our hazard.

Control Hazard Solved by Delay Slot



Alternatively, we can **modify the definition of the architecture** and decide that **the two instructions following a branch are executed in any case** (branch taken or not) as if they were before (same thing as in cs-328). Thse instructions after the branches are called **delay slots**, before when a branch happened we added `nop`, so might as well try to compute someting instead right? But now our code becomes **counterintuitive**, we execute code maybe for nothing.

Remark MIPS did it as some others, but quite rare in current architectures

Use of Delay Slots

A simple way of using delay slots is to use them for `nop`'s– but then it is not better than stalling the pipeline. A better idea is to put there instructions which precede the branch and on which the branch has **no dependence**. Suppose an architecture with two delay slots:

```
sub r2, r0, r7
mul r1, r6, r7
add r5, r3, r4
beq r0, r1, loop
nop
nop
lw r8, 12(r9)
```

Then this can be turned into:

```
mul r1, r6, r7
beq r0, r1, loop
sub r2, r0, r7
add r5, r3, r4
lw r8, 12(r9)
```

We use some instructions that is independent to fill the nop instructions

Branch Prediction

- A better strategy is to **guess the branch outcome** and fetch the corresponding instruction (either the next instruction or the branch destination, but not nessarily the former)
  - If the guess is correct, **no cycle is lost**
  - If the guess is wrong, what has been fetched and decoded is thrown away (**squashed**)
- Branch predictors of modern processors are extremly sophisticated: dynamic predictors **learn from previous executions** of branch...
- Complex predicots can be **correct up to 95-99%** of the time
- The quality of branch predictors has made architectures with delay slots extremly rare

Three Types of Hazards Hinder Pipelining

Solutions

Data Hazards	• Forwarding paths, wherever possible • Stalls in all other cases
Control Hazards	• Delay slots, if the architecture allows it • Branch prediction, to try to do the right thing • Stalls, if not

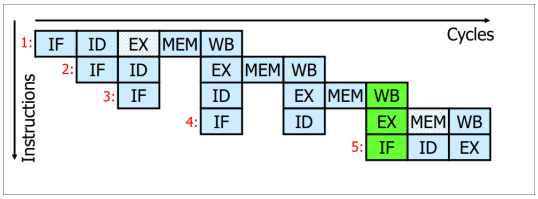
Structural Hazards

- Rigid pipelines which cannot have structural hazards by construction
- Stalls, otherwise

4.4 Dynamic Scheduling

Starting point

As we have seen before, processor has first been **sequential multicycle processor**. Now we wanted to replace it with something else: **pipelined processor**.



For instance, 5-stage pipeline with all forwarding paths. Typical of **MIPS** and **RISC-V**. The issue with this is that **all instructions** has to go through **all the stages** $\implies$ we need a very small ISA (MIPS, RISC-V). The other issue is that pipelined need **independence**. Every circuits in each stage has to be used **only in that stage**. This is the simplest form of **Instruction Level Parallelism** (ILP): several instructions are now executed at once.

Simple Pipelining

The scope if **parallelism is limited**:

- **Data hazards** limit the usability of the pipeline:
  - Whenever the next instruction cannot be executed, the pipeline is stalled and no new useful work is done until the 'problem' is solved (e.g., cache miss)
- **Control hazards** limit the usability of the pipeline
  - Must squash fetched and decoded instruction followig a branch

Rigid Sequencing

- Special 'slots' for everything even if sometimes useless (e.g. M)
- Every instruction must be **coerced to the same framework** (floating point vs. integer)
- Structural hazards avoided 'by construction'

This means that if we need some floating point arithmetic then we are a bit lost on our RISC processor. We would need to implement the floating point in software (which is slower).

Dynamic Scheduling: The Idea Let us for instance take a look at the code:

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

Here the instruction `divd` is a long-running instruction which hurt our feeling...it makes our program **stalls** for the next instruction `add f10, f0, f8` which need the value `f0`. However as we can see the instruction `subd` is completely independent of the two above. As we have seen before, maybe a way of resolving the stalling is by using a 'smart' compiler which would put the `subd` instruction between those two. However here our `divd` takes a long time but how much? how many cycles? We cannot know right? Imagine having a `load` instruction we wouldn't even know!

Théorème 2. Relax a fundamental rule: instructions can be executed **out of program order** (but the result must still be **correct**).

Break the Rigidity of the Basic Pipelining

For us we need:

- **Continue fetching and decoding** even and especially if one cannot exevcute previous instructions
- **Keep writeback waiting** if there is a structural hazard, without slowing down execution.

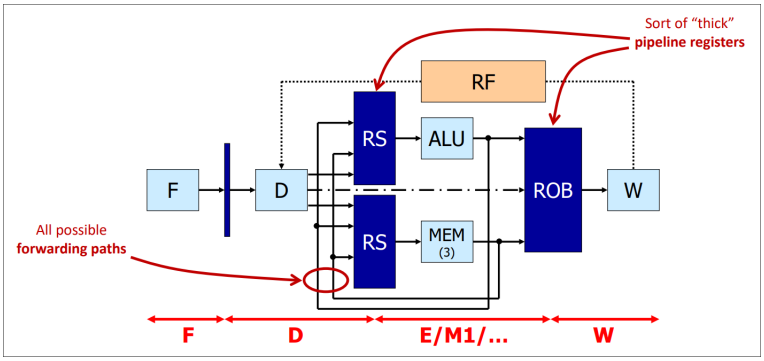
Solution

The solution here is the **splits the tasks** in independent units/pipelines

- **Fetch and Decode**
- **Execute**
- **Writeback**

Dynamically Scheduled Processor

So now the we don't have to go through one big pipeline. Now we break our previous big pipeline into smaller pipeline. We may have a big pipeline for floating point double and a smaller one for 32 bits floating point.



So what is going here is we create a big Execution unit with all the possible instruction (memory, alu, etc.). We then wire them back into our decode just as we did before. Now everything that produce a result we bring it back **where** there is a possible use of it.

Remark 41. Here what we have is still a finite state machine. The reason why we don't present it as one is that the number of state that is can have is enormous. It the a combinatorial calculation between all the state F, D, E/M1, .., W.

Problems to solve

Now we have a kind of big work to do, all the previous issues that we had already solved come back again at us here: **Structural Hazards**

- Are the required resources available?
- New problem: previously handled by rigid pipeline

RAW Data Hazards

- Are the operands ready to start execution
- Old problem

WAR and WAW Data Hazards

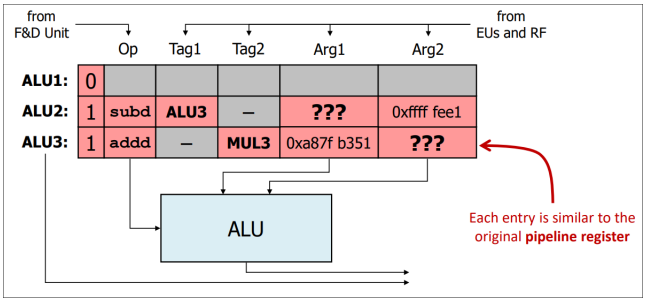
- The new data overwrite something which is still requires?
- WAW is a completely new problen – impossible before, WAR often cannot occur

Raw Data Hazards

Let us start with this one. This problem comes fron the `RS` register from the previous picture. Maybe what we read the register and the value from it is false. We want to erase it. The solution here is to use a **Reservation Station**

Reservation Station

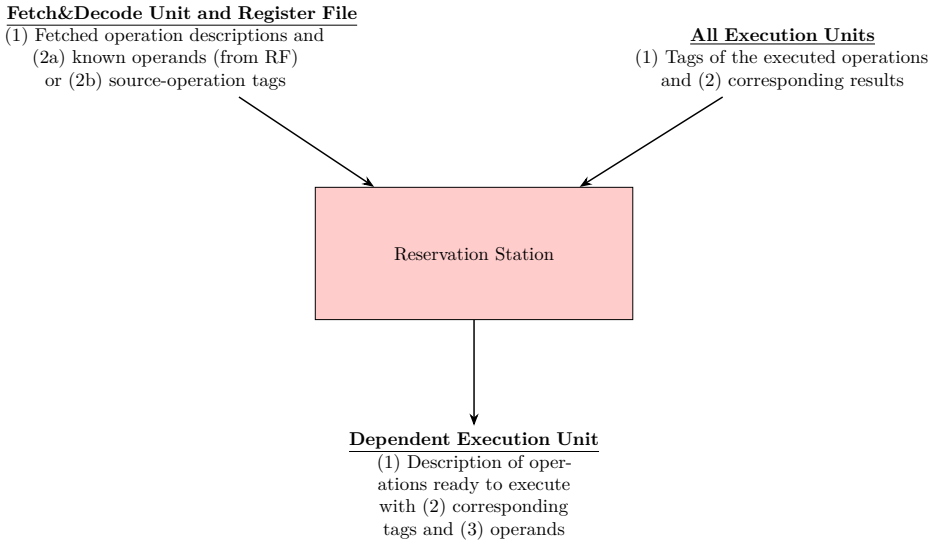
Definition 8 (Reservation station)**.** checks that the **operands are available** (RAW) and that the **Execution Unit is free** (Structural Hazards), then starts execution.



So here for instance the `ALU3` means that the value we are waiting for will be the result of the `ALU3` instruction im the reservation stations.

The parking lot We can see our Reservation stations as a **parking lot** instead of having car we have instructions waiting to be ready. Everytime one is ready then we launch it. This is a completely **unordered** 'data structure'. On the bottom we have a ALU that is ready to compute

something **every cycles**. On the top then we have all the come back pipe which comes to us and tell for intsnace 'I am the result of the MUL3 computation' we then ask ourself if any of us (instructions) is waiting for the `MUL3` result.



What is a Reservation Station?

A **reservation station** is a small buffer inside the processor that holds instructions after they have been decoded but **before** they are executed. Its purpose is to allow the CPU to execute instructions **out of order** without violating the correctness of the program.

A reservation station entry contains:

- the **operation** to execute (e.g. `add`, `mul`),
- the **operands**, if they are already available,
- or **tags** indicating that the operand is not available yet,
- and the **destination tag** where the result must later be written.

The idea is simple: an instruction does **not** need to wait for its operands in the decode stage. Instead, it is placed in a reservation station entry, even if some operands are missing. The entry then "listens" to all execution units. Whenever a unit finishes and broadcasts a result together with its tag, each reservation station checks whether it was waiting for that value.

If an entry collects **all** its operands, then that instruction is marked as **ready**. As soon as the corresponding execution unit becomes free, the reservation station dispatches the instruction to it.

In short:

- It solves **RAW hazards** by waiting for the true operands.
- It helps avoid **structural hazards** by checking that the execution unit is free.
- It enables **out-of-order execution**: instructions no longer need to wait in program order if they already have what they need.

A reservation station therefore acts as a **smart waiting room** for instructions: they enter as soon as they are decoded, wait only for the data they require, and are issued to execution as soon as they are ready.

So here our read after write issue is gone. But the WAR or WAW are not resolved. The question the Prof. asked was:

> Why don't we use something simpler for our tag (e.g., the register name)?

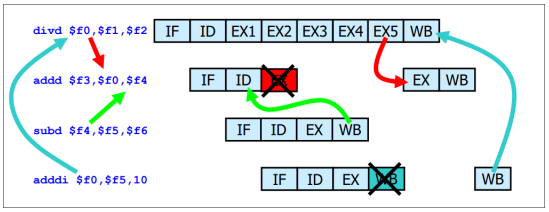
Our register are not unique, this would work only if we were in an ordered program where each instruction are followed by the next one. If that was the case then yes this would work because as the register t4 we would have the latest value of t4 which is the one we need. But what happens if we are in an unordered instruction table (Reservation stations)? Then in that case it could be possible that there is a previous value that is not the correct one in t4, t4 would not be unique.

We need **unicity** we have to be sure that when calling a tag then the value of this tag **has to be unique**.

So why don't we use the program counter as a tag? By using it then we have a unique identifier for each value right? But what if we have a loop, then for the same pc we would still have two different values.

WAW and WAR

Those hazard were not really possible in a simple pipeline (by construction). However here this is different, for instance:



So here we have a new issue. The solution for this would be for instance to rename our register like: Given this code

```
divd f0, f1, f2
addd f3, f0, f4
subd f4, f5, f6  #create issue with the f4 of the previous instruction
adddi f0, f4, 10 #create issue with the f4 of the previous instruction
```

We can change it to:

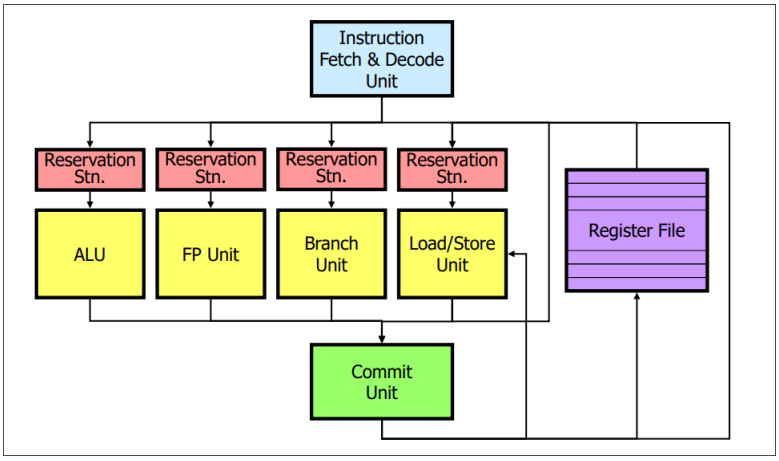
```
divd f0, f1, f2
add f3, f0, f4
subd f30, f5, f6
f29, f30, 10
```

But here our reservation station comes down pretty much as renaming each register as the tag.

- Unavailable operands are identified by the **name of the reservation station** in charge of the originating instruction
- **Implicit register renaming**, thus removing WAR and WAW hazards
- New results are seen at their inputs through special result bus(es)
- Writeback into the registers can be in-order or, to some extent, out-of-order

4.4.1 Dynamically Scheduled Processor

So here what we get at the end is:



What we here as an issue still is how do we know where to write in the register file, we need to be careful about our ordering in our register file. This is the commit unit's job. **Our-of-order**

Commitment and Exceptions

- Excpetion handlers should know exaclty where a problem has occured, especially for **nonterminating exceptions** (e.g., page fault)
- Of course, one assumes that everything before the faulty instruction was executed and everything after was not
- With dynamic execution it might no longer be true...

Problems with exceptions

Now we have an issue again with excpetions:

Precise exceptions

Reordering at commit; user view is that of a fully in order processor

```
andi t4, t2, 0xff # good code
andi t5, t4, 0xff # good code
addi v0m t5, 1  # good code
srl t2, t2, 8 # good code
-> lw t3, 8(t6) # now we have our exception
andi t4, t3, 3  # bad code
addi t0, t0, 4 # bad code
addi t1, t1, 4 # bad code
```

So now we need to come back from where were at the exception. To do so we need to save the pc of where the exception has happened we already have it and returning to it is already implemented (the `ret` in RISC-V).

Imprecise exceptions

- No reordering; out-of-order completion visible to the user
- The OS/programmer must be aware of the problem and take appropriate action (e.g., execute again the complete subroutine where the problem occured)

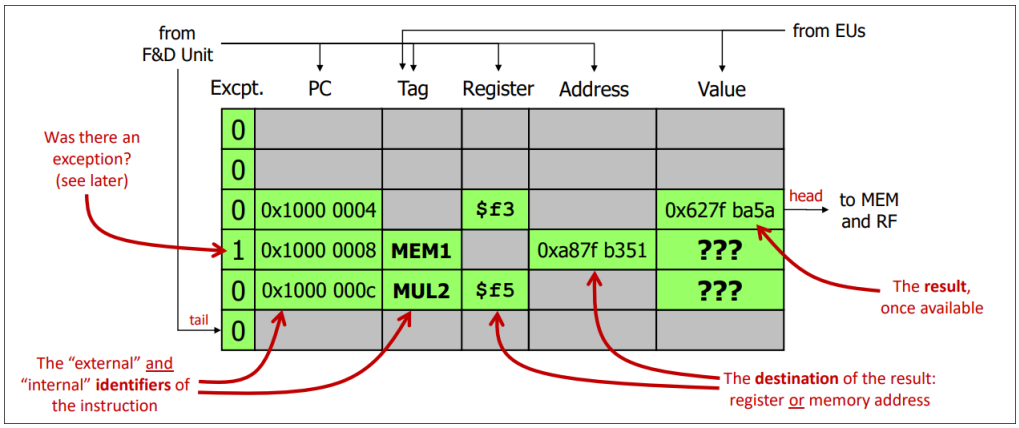
```
andi t4, t2, 0xff # good code
andi t5, t4, 0xff # bad code
```

```
addi v0m t5, 1  # bad code
srl t2, t2, 8 # good code
-> lw t3, 8(t6) # now we have our exception
andi t4, t3, 3  # bad code
addi t0, t0, 4 # good code
addi t1, t1, 4 # bad code
```

Now here we are not able to jump again, if we were, yes our program would run again fine but then that addi here has already been computed (`addi t0, t0, 4`) would be refetched again which would make our `to` incremented two times!

Reorder Buffer

The way of solved this issue is to add a new buffer between the output and the register file:

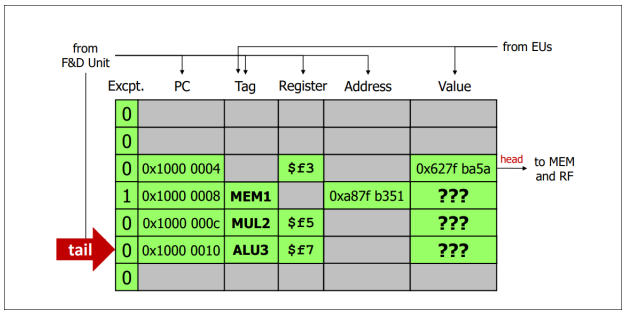


This one is pretty different from the other buffer. This one **has to be ordered**. This buffer works kind of like our pipeline registers.
Now this buffer **has to know where the instruction where**. And it also need to know where, what who, etc. So here this is not like the previous parking lot, the order now is important. The purpose of this unit is to make the user believe that everything is sequential.

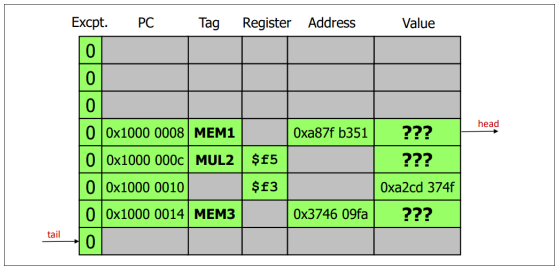
Remark 42. For this part (the example the video is always better than this so this is *cs-200 – 4d. Instruction Level Parallelism (cont'd) around 20:00*) But I will still try:

Example

Let us take for example the following reorder table. As we can see with have an head and a tail. The head represent the element which has the smallest `pc`.



So firstly we check the first lane that is at the head: `0|0x100 0004| |$ f3| |0x627fba5a`. As we can see ther eis no tag or anything that we need to wait for → we can commit the instruction!



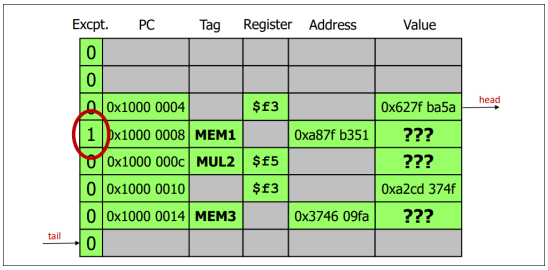
Now this is different: we currenlty don't have the value. We need to wait until the oldest instruction here has its result. As soon has it has the value, then we can write to the memory the value.

Reordering and Precise Exceptions

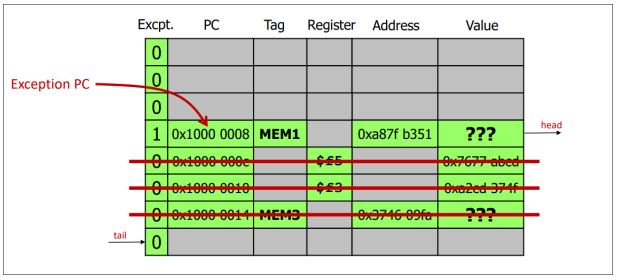
However here we still have the big issue of **exceptions**:
So we need some extra bits in the reorder buffer to know wehter or not there is an exception on a given instruction. Okay this is nice now we know when an instruction happens but what should we do?
So now we can always execute the previous instruction without having to worry about anything right? But what happens when the head is on the exception pc?

- When a synchronous exceptions happens, we do no report it but we **mark the entry** corresponding to the instruction which caused the exception in the ROB
- When we would be ready to **commit** the instruction, we **raise the exception** instead
- We also **trash** the content of the ROB and all RSs

Example



So here what happened here is that we tried to store the value out of the **MEM1** unit which result in a TLB miss → we simply **record the exception**. Now we can still execute the instruction that is at the address `0x1000 0004` as nothing has happenend.



Now that we are on the current instruction, we **need to actually to notify** the world about our problem. We can now raise out tlb Miss exception. But all the thing that we had computed before is rubbish, we cannot know if those value are actually correct or just something that were computed with the wrong values, to solve this issues, we just **erase** everything that were in the reorder buffer.

Problem we need to solve?

The structural hazards were not an issue here, the **reservation stations** take care of the contention for execution units. The **commit unit** writes back one instruction at a time.
But what about RAW Data hazards, but those are taken care of by the **reservation stations**

Reservation stations

We said as before that: as the decode unit, we get an instruction, if the input are available then we put the arguement and the op etc...of the instruction in the reservation stations, else, we put the tag of the instruction...

How do we know?

This is the decode unit problem, the decode unit has to know if the parameter is available, but how can he know. He receive an instruction like "add the register x2 and the register x4" but how can he knows if he can read the value or not?
Before, what we used to do is to check in the past. In a simple pipeline processor the 'past' is actually on the 'right' of the pipeline, we can check if any instruction on the right of the pipeline has as output one of the register we currenlty have, if yes then we know that we have to wait. Can we do something like this now?

Decoding and Dependence

The **reorder buffer (ROB)** knows of all instructions not yet committed and of their desitnation registers

Possible situations

- **No dependence** → Read the **value** from the `RF`
- **Dependence** from an ongoing instruction
  - If the value is already computed → Get the **value** from the **ROB**
  - If the value is not yet computed → Get the **tag** from the **ROB**

Dependences through memory

The way we detect and resolve dependences through memory (a store at some address and a subsequent load from the same address) is the same as for registers
For every load, check the ROB:

- If there is **no store to the same address** in the ROB, get the value from memory (i.e., from the cache)
- If there is a **store to the same address** in the ROB, either get the value (if ready) or the tag

But there is an additional situation now

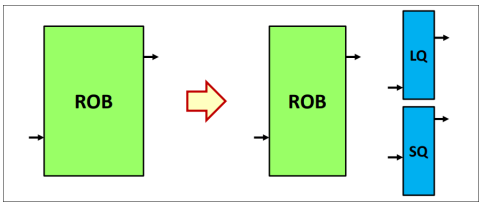
- If there is a store to an **unknown address** in the ROB or if the address of the load is unknown, **wait**

Remark 43. The memory and the register file comes down to the same thing, conceptually we have to do the same thing for both.
The slide difficulty for the implementation of the memory is: when we are looking for a certain register we only have to check ('is there an `f1` here?'). When we wont to write something, it can happen that the address is not ready yet. So when we want to read at a certain address, but there were a write before at an unknown address, we just cannot know if the value at the certain address will be overwrite or not?

The solution for this is to use a Load store queue:

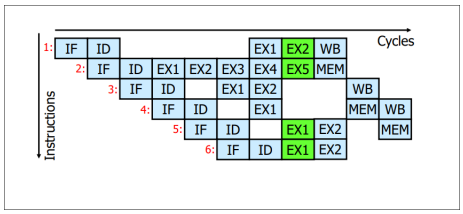
Load-Store Queue

In practice, the **memory part of the ROB** is implemented separately and is called a **Load-Store Queue** (in turn, usually implemented as a Load and a Store queues)



Second Step: Dynamic Scheduling

So here this is where we are at:



We gain to ability to reorder instructions → **unlocks a tangible amout of ILP**

Is it easy to implement?

The question the professor asked is:

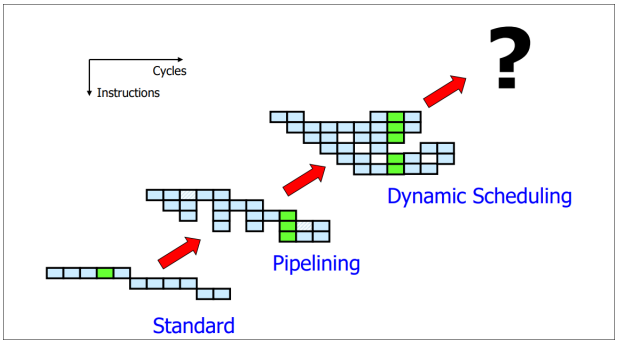
Is it easy to implement (in verilog for example)

Let's take for instance the Reservation Stations, How do we size the table:

- If we take a smaller size, it block → we are dead
- The second level of problem is: what is the best number? Taking 32 or 16 maybe doesn't change the result but which one is the best for us?

The way of doing this is experiment right? we can try to simulate our processor to see which size is the best!
But at the end the reservation stations maybe just be a piece of memory right? But The sad thing here is: we have to search everytime we received a new element then we have to check every element if they care or not. Also, we have **all the data buses** such as the F and D unit, the EUs and RF which gives information to our reservation stations. all of those big buses has to be 'concentrated' into a couple of flipflop. We have the same issue for the reorder buffer, we have also to search in our big buffer

Summary: ILP So Far



How we have seen the reorder buffer and the reservation stations, with the tag that is stored inside the buffer is a pretty naive way of seeing it. Yes it works, but is it practical? Not really

4.5 Scheduling Examples

The is the program that we will want to run:

```
lw x1, 0(x5)
addi x5, x1, 1
lw x1, 0(x6)
add x3, x8, x5
subi x2, x6, 1
subi x4, x3, 5
add x3, x2, x4
lw x2, 0(x7)
or x4, x2, x1
subi x7, x3, 9
```

Goal

The goal for us with this section will be:

- Determine the **execution schedule** for this code in five different architectures
- **Compare the number of cycles** required by the five architectures
- **Compute the CPI** (and IPC) of all architectures on this program

4.5.1 Architecture 1

For this first architecture, we will use a **multicycles** processor without any pipelining. The execution latencies are the following:

- ALU operations → 4 cycles
- Memory operations → 6 cycles

So for us here this is very simple to count the number of instructions so here there are three load and 7 ALU operations:

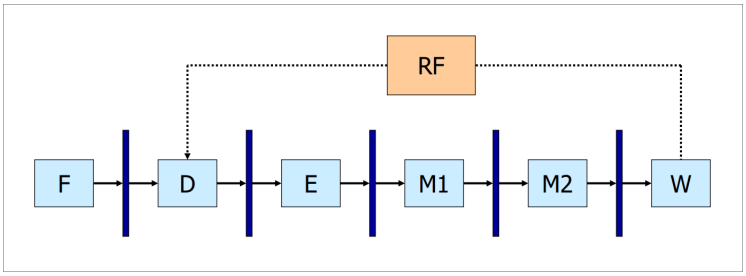
$$3 \cdot 6 + 7 \cdot 4 = 46$$

What we also want to compute here is the average cycles per instructions:

$$\frac{46}{10} = 4.6$$

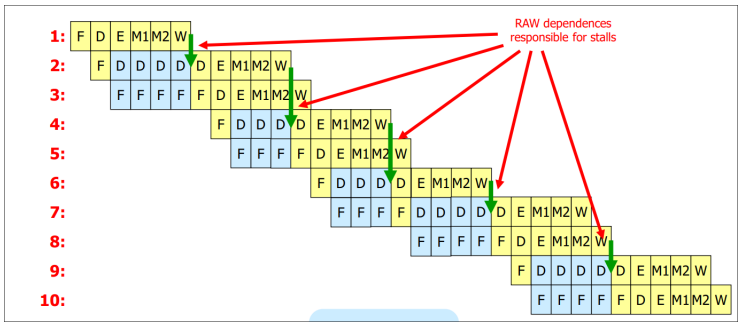
4.5.2 Architecture 2

Here we will have a **6-stage pipelined** processor without any forwarding paths



So here we have two cycles more that every **non load/store instructions** has to go through. It can be even worse if we have dependency.

The way of doing it is step be steps: first the first instruction → it doesn't depend on anything so we can just put without any dependency. Then we check the second instruction if it has any dependency → it has one: the `x1` this means that we will need to wait until the writeback for the next instruction is done. we will need to stall our processor four times. Because this is a rigid pipeline, everything in our processor is stalling so we need to fetch again and again our value (even if it is correct) until our CPU is again free. So for the second instructions, we have fetched 5 times the correct value.
At the end here if we continue until the end of the program we get the following:



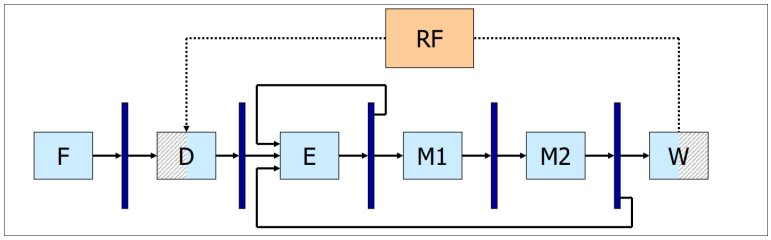
We needed **33** cycles →

$$\text{CPI} = \frac{33}{10} = 3.3$$

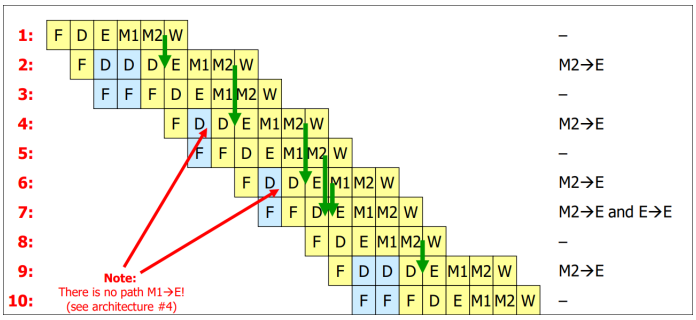
4.5.3 Architecture 3

Now we still have a **6-stage pipelined** with **some forwarding paths**

- E → E, M2 → E
- W → D



Let us take the first instruction `lw x1, 0(x5)`. The result of this instruction will be available after M2 right? which means that as soon as M2 has finished his job → we have that the result is direclty transfered into the Execution stage (M2 → E). In comparaison of what we did before; we gain **one whole** cycle.



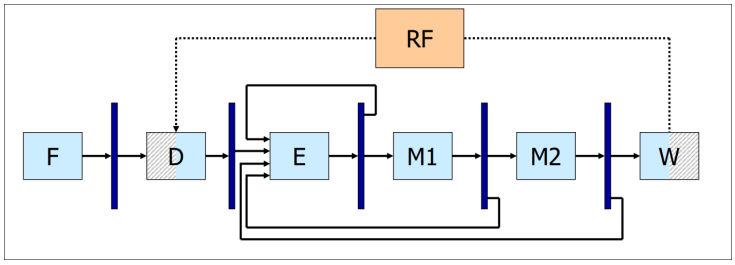
If we count the number of cycles needed we get 21:

$$\text{CPI} = \frac{21}{10} = 2.1$$

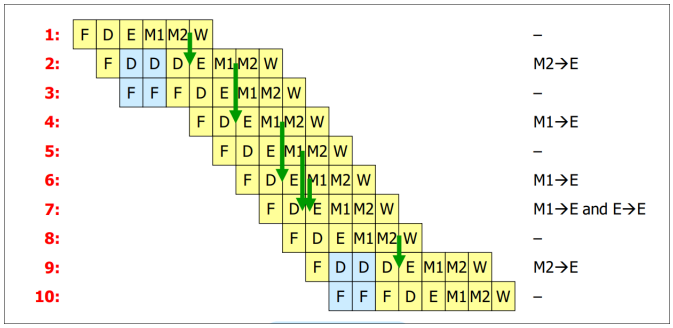
```
1: lw x1, 0(x5)
2: addi x5, x1, 1
3: lw x1, 0(x6)
4: add x3, x8, x5
5: subi x2, x6, 1
6: subi x4, x3, 5
7: add x3, x2, x4
8: lw x2, 0(x7)
9: or x4, x2, x1
10: subi x7, x3, 9
```

4.5.4 Architecture 4

We still have a pipelined processor (the last one), but here we have **all the forwarding paths**.



Here the first instruction doesn't change from what we did with the third processor. The fourth and the second instruction has a dependency, but this times instead of having to stall the processors we can actually retrieves our value from M1 which makes us gain one cycle. If we following the same principle for all the execution we get:



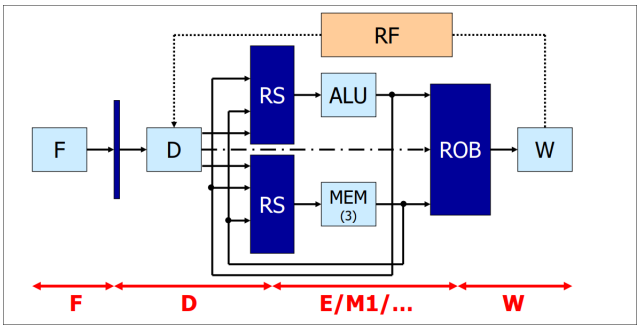
If we count the number of cycle needed we get 19 which is very good:

$$\text{CPI} = 1.9$$

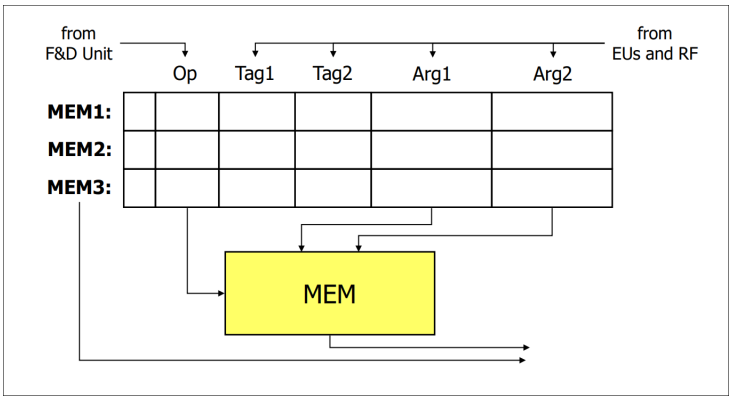
4.5.5 Architecture 5

Now we have a **Dynamically scheduled**, out-of-order (OOO), unlimited RS and ROB size

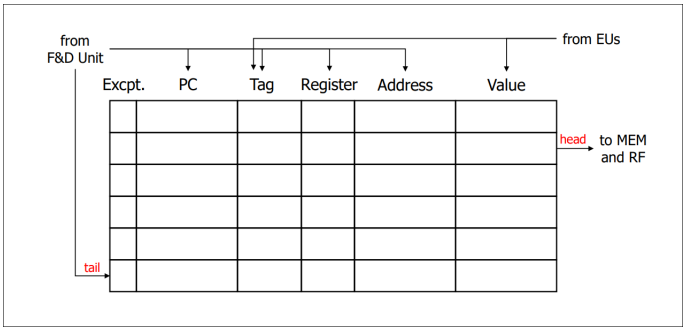
- 1 ALU (latency 1) + 1 Memory Unit (latency 3)



So we also need a Reservation Station, one for each execution Unit, in our context, we have two executions unit



We also need a reordering Buffer (ROB)



'to do list' for simulating Dynamically Scheduled Processor

- At the beginning of each cycle
  - **E phase** – Issue ready instructions:
    - From all RSs, issue as many ready instructions as there are FUs available
  - **W phase** – Writeback results in order:
    - Remove top entry from ROB if completed (**ONLY THE TOP**)
- At the end of each cycle
  - **D phase** – Load result of decoding state:
    - To the relevant RS, including ready register values
    - To the ROB, to prepare the placeholder for the result
  - **E phase** – Broadcast results from all FUs:
    - To all RSs (incl.deallocation of the entry)
    - To all the ROB

Remark 44. The explanation on how to run this is 14:00 in *CS-200 – 4e. Instruction Level Parallelism (cont'd)* from the 4th December

So here we have the first two cycles which are the same as the usual pipelined processor, the. We have decoded the first instruction which is a load words with the arguments 0 and 555. So we then put the instruction into the reservation table for the memory unit at for instance **MEM2**.

		Op	Tag1	Tag2	Arg1	Arg2
MEM1						
MEM2	e	lw	-	-	0	555
MEM3						

Remark 45. The order or where we put it in the reservation table doesn't matter

The decodin also need to keep track of the fact that this instruction exists → is also put the information in the reorder buffer. We put the instruction at the top of the ROB with the following informations:

- PC = 1
- EX = 0
- At the current time, the tag we are using for this instructions **must be unique** here MEM2 **has to be unique**
- so the register is `x1`
- The address is nothing because we don't have to write to the address
- We don't know the value at the current value

We do now to the next cycle: we check if there is any tag in our MEM2 station, there is none (obviously because this is the first instruction) therefore we can enter into the memory unit. Has we have said before it has a latency of three which means that this instruction is gone for three cycles (there is no going back).

Should we erase this instruction from the reservation table? I mean we don't use it now so why should we keep it?

But if we let someone take our parking spot, then MEM2 becomes the new instruction right?

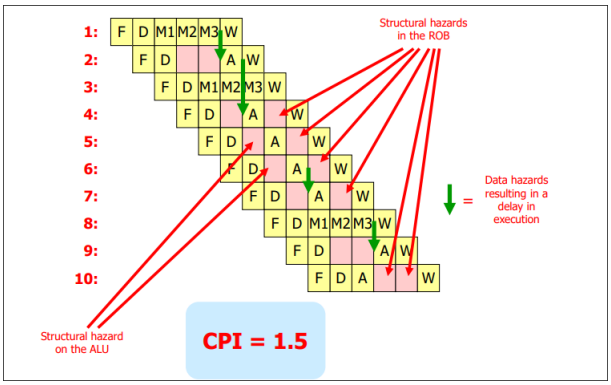
But we are not done with the previous MEM2, so we would have a mismatch of MEM2.

We need to add a bit of 'state' to know what is up with our tag. the bit would be 1 if we are currenlty executing the instruction else 0.

At the same time the second instruction also is decoded at the second cycle. so this is an immediate instruction → we only need to check one register `x1`. We check the ROB if we see that there is

`x1` then we know that it is not ready. As we can see there is `x1` → we check tat tag MEM2 and we store this tag in our reservation station of the ALU unit. we have for instance

		Op	Tag1	Tag2	Arg1	Arg2
ALU1						
ALU2						
ALU3		add	MEM2	-	-	1



Solution

This is the ALU rs

		Op	Tag1	Tag2	Arg1	Arg2
ALU1						
ALU2	w	add	-	-	257	93
ALU3		sub	-	-	456	1

And this is the MEM rs

		Op	Tag1	Tag2	Arg1	Arg2
MEM1						
MEM2	e	lw	-	-	456	0
MEM3						

This is the ROB

Excpt.	PC	Tag	Register	Address	Value
0					
0	2	-	x5	-	93
0	3	MEM2	x1	-	???
0	4	ALU2	x3	-	???
0	5	ALU3	x2	-	???

4.6 Besides and Beyond Superscalars

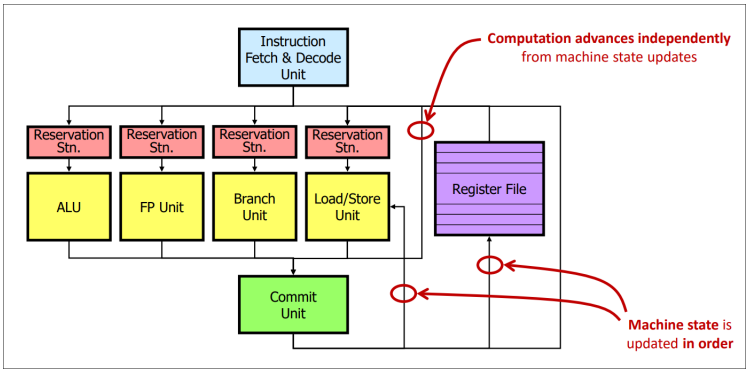
So far we used to way to optimize our processor: pipelined and dynamic scheduling, for this lecture we will see if we can go a completely different way of what we have done for us:

Content of this lecture

- Superscalar processors
- Speculative execution
- Simultaneous multithreading
- Nonblocking caches
- Very long instruction word (VLIW) processors

To this day

So for us our processor look like this:



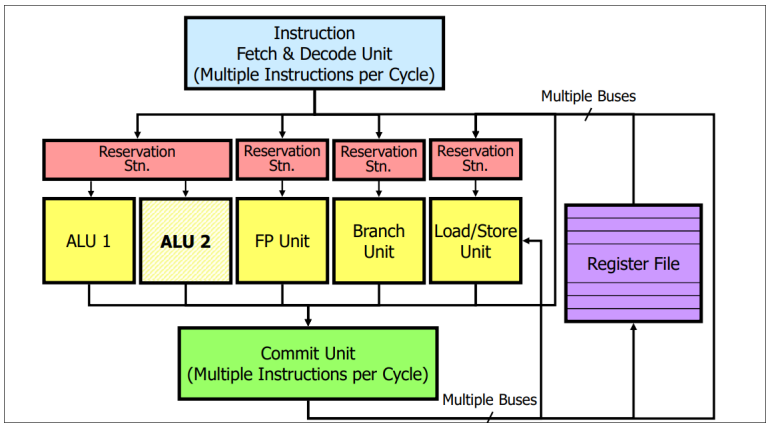
We have two big **seperate parts** in our circuit:

- The ordered part: between the commit unit and the registerfile, instruction Fetch and decode unit
- The unordered part: the reservation stations and all the unit

The limits we have is for instance imagine we have 3 alu instruction that are ready, then we have to execute them sequentially which 'slow down' our processor. The second limitation is that maybe we don't have enough instruction in our reservation station.

Superscalar Execution

The solution for this for instance is to add a new ALU execution unit:



And this is pretty easy to do, instead of looking for one instruction per cycle (as the ALU execution unit), we can look for two instructions per cycle. But this may be not that great right? maybe because of a lot of depencies etc. we don't really gain any time with this principle.
A solution to did is instead of fetching one instruction at a time (in the Fetch and Decode unit), we can fetch two? Therefore committing two instructions per cycles (maximum). Is it easy to build? if we just copy paste the Decode unit, then we can decode two unit per cycle right?

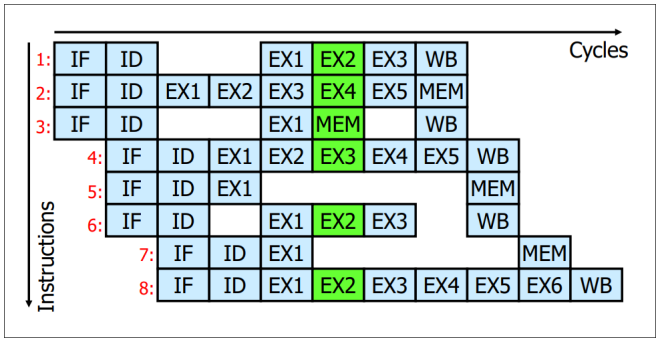
Remark 46. What about depencies? What if the second instruction actually depends on the first ones, then searching in the commit unit is not sufficient anymore. We will also need to take a look at the other decode unit to check any depencies.

- **Fetch more instruction per cycle** no big dufficulty if the instruction cache can sustain the bandwith
- **Commit more instruction per cycle**: The ROB and the register file must have enough ports
- Obey data and control depencies: dynamic scheduling already takes care of this

Remark 47. **Data and control hazards** are the **ultimate limit** to **parallelism**

Superscalar Execution

If we take an execution of a code this would be the graph:



Which one was first

Superscalar was not really created after dynamic scheduling. In fact in the 90s we were already doing superscalar processor before adding dynamic scheduling.
For instance we had two different pipelines: one for integer and one for floating point. For each fetching, we take two instructions; we hope that one of them is an integer and the other is a floating point, if it is: **jackpot** we put them simultaneously in the pipelines → we gained one cycle. If they are both of the same types then we just forget about the second one and do as we used to do in a classical pipeline!

Intel Processor and Fetch

In the late 70s instruction were not fixed size. This means that some instructions were one byte long, other were 50 bytes long. But why did they do that? At that time memory was a big limitation, imagine if you had 1000 bytes of storage for your program, then you need to compress you code as much as possible → making the most common instruction the smallest and the least common instructions the biggest. This is a very good idea... at **that** time. But now memory is not a big issue anymore.

Remark 48. How does it works: The way of decoding instruction is kind of like a stack: while the instruction is not done → we add the next byte to the instruction. (adding the next byte wasn't an issue because the memory was byte addressable at that time)

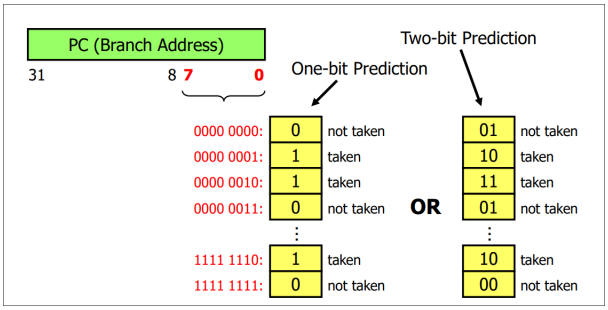
Imagine being with instruction that have not a fixed size and that we want to make some decoding in parallel... we are cooked! There is no way for us to know when our instruction stop before actually decoding it.

4.6.1 Dynamic Branch Prediction

All the thing we have said before is nice if and **only** if we know where the next instruction is before executing the current now. But this is not always true...

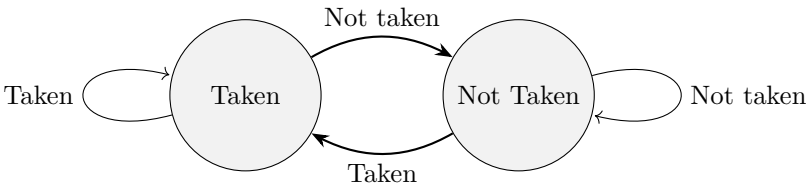
- The **biggest problem** left to continue extracting instruction level parallelism are:
  - **True data depencies**: instruction **cannot** be executed! Not much we can do about...
  - **Branches**: where to look for other candidate instructions?
- **Static** prediction not very accurate and somehow hard to use
  - Never-taken, Always-taken-backward, Compiler-Specified
  - How does one know which one is right?
- **Dynamic** prediction: learn from history
  - Count how ofter a branch was taken in the past

People tried to predict the branch output, if we know that the branch is usually taken or not, then we can have a better prediction than usual, but how us as the compiler would even know about that?

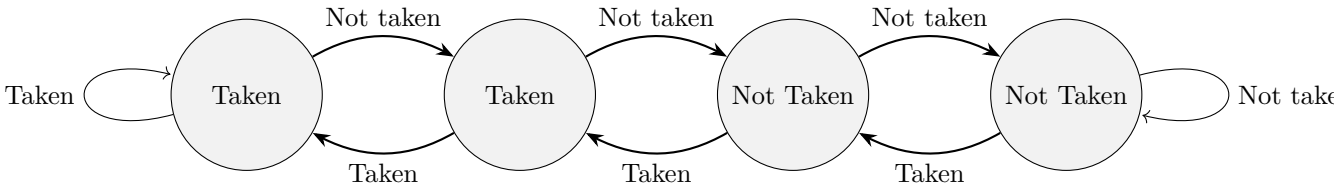


So the prediction's job is given directly to the processor. To do so, it will hash the address of the branch instruction with the taken or not taken information (maybe they will be some overwritting in the table, we don't really care because this is just a prediction at the end so being wrong is not a big deal).
This is kind of easy to do right? we just have to create a big table where we store each branch that we cam through. In fact people do usually something a big more complicated, we use a finite state machine: **One- vs. Two-Bit Prediction Schemes**

The simplest one is a one-bit predictor which is basically a 'do the same as last time':



A two bit predictor (saturating counter): adding some 'inertia' or 'take some time to change you mind'

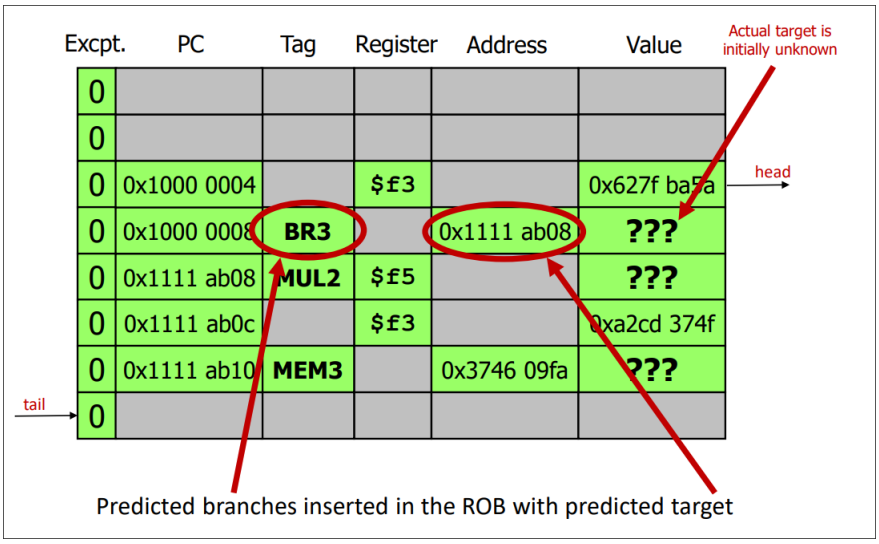


So now we can use this guess to try to fetch and decode the right instructions. The question we have is: Can we go further with this, can we actually execute the instructions: **Speculative**

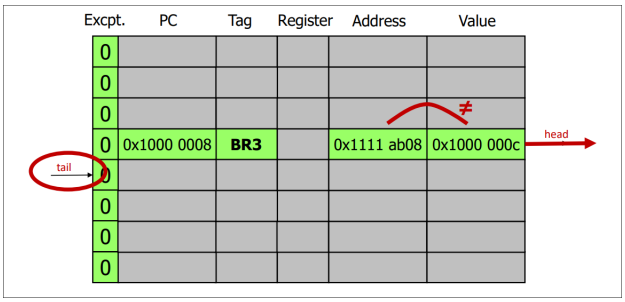
Execution

- We have been using **Dynamic Branch Prediction** only to tentatively **Fetch** and **Decode** instruction → no effect on registers and memory, so **easy to squash**
- More aggressively, one could **Execute** instructions (and use their results) before the branch targert is known: **Speculative Execution**
- We need to **prevent changes to the architectural state** of the processor until the correctness of the prediction is known:
  - Was it right? Good!
  - Was it wrong? **Squash it**

So here after executing the instruction instead of waiting for a value as we did before, we are actually waiting for our branch result to know wether or not we are cottect. This means, the value of our instruction is unknown **and** that we are waiting for the branch tag to come up ((BR3 for instance))



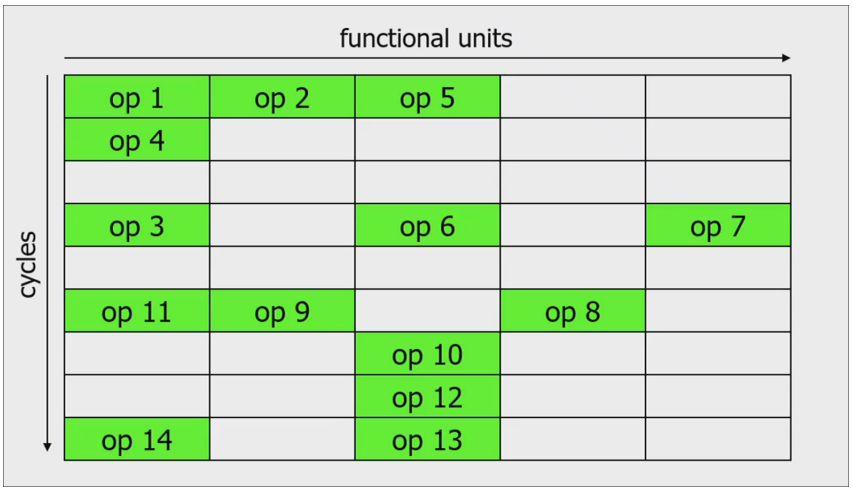
If we were wrong in our prediction then we do the **exact same thing** as exception. So here we must wait until the **BR3** is known. If we were correct then we can commit the value `0x10000 0008` instruction **and** all the next instruction are already computed for us!
But what happens if we are wrong then, we need to **squash** all the next commit:



A mispredicted branch triggers a squash. As we say in french 'ni vu ni connu' (*Prof. Ienne*)

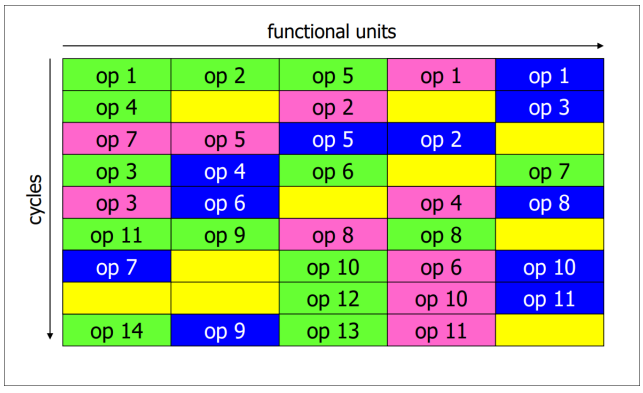
4.6.2 Simultaneous Multithreading

Before explaining what's multithreading, first let us see what our current processor execution looks like:



So here we can see that there is a lot of unit that are not used at the current time. Each blank boxes here I could use it for something that is useful for us, they are free.
How can I make a profit based on that? If we remember how our computer works especially with the operating system, there is always a lot of program that are running at the same time. At the current moment we were 'seperating' in time slots, each program run for like 0.01 second and then another program run for 0.01 seconds, etc. But why don't we just run the second program instructions in the place which is blank here? (let us first think about the things that would work properly and the other thing that would'nt work).First **dependencies** there is no dependencies between the first program and the second program which is great. **But** there is some big issues about the registers. the first program use 32 register, but the second programs also use 32 registers, and those registers (the ones of the first program and the one of the second program) **cannot** be the same. We will need to add a registers for everyone → two sets of register.
A another **big issue** is the **PC**! we also need two PCs
Another issue is memory, how do we access memory for both, the OS helps me go to memory in the right way, but for us now, we need two way to do so which means that the TLB need to have something. We need to have the ability to translate the two different access to memory at the same time.

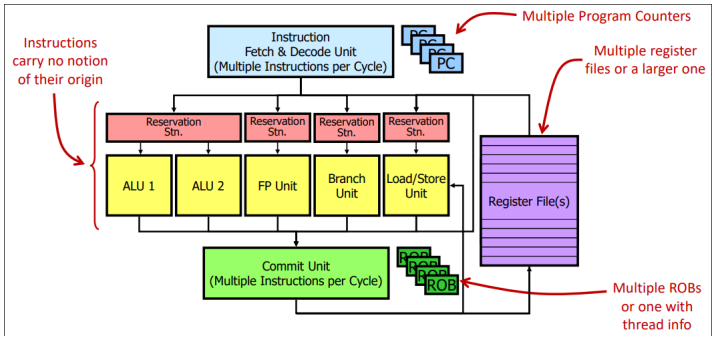
After adding this for two programs, why not three, four, etc.. The goal for us would be to obtain something like this (having most of the space filled):



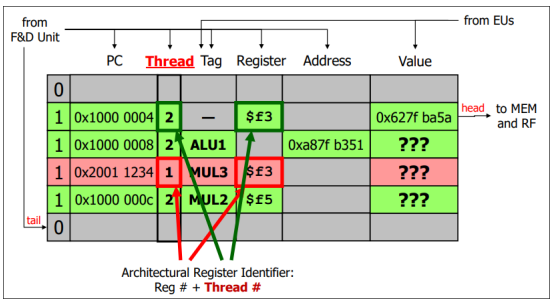
When accessing in memory, we need to have the information "this is a memory address for the pink", "this is a memory address for the blue", etc...

How do we do it?

First we'll need to add multiple PCs (as we said before) → Mutliples ROBs or one with a thread info. that's it?
First let us remember how our instruction are implemented/used in the execution unit (from the reservation stations to the commit unit) At those place there is a **total abstraction** between the **program** and the **instruction**. What this means is that: in the reservation stations. The instructions that are being stored doesn't know from where they come from. They are just there chilling waiting to be executed. They have different names waiting for tag and not instructions etc... So for us this is perfect! we don't have a lot of work to do because of this abstraction. We need to have either a big register files, or multiples.

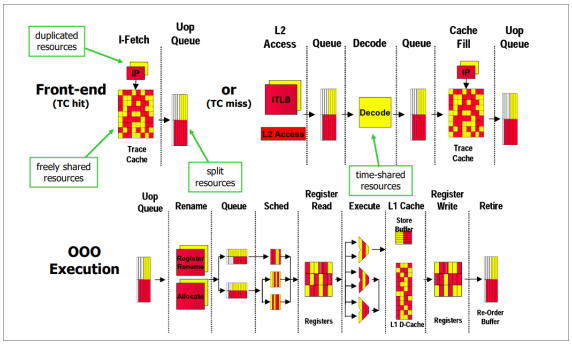


The only guy who knows that we are multithreading is the **reorder buffer**, it remembers the thread of origin of each instructions.



Intel SMT: Xeon Hyper-Threading Pipeline

Let us look a bit in the past. If we take a real example, the first processor that implemented this idea:



This is the first **real** pipeline that we are looking. The first thing we can notice is that there are more stage in this pipeline as our 5-stage pipelined ones. One of the reason for this is that intel at that time was 'racing' to the fastest processor.

Remark 49. Intel has the tendency to renames everything: Hyper-Threading instead of multithreading, IP instead of PC, etc.
Remember to look at the schema and try to understand each part of the pipeline

What do we loose to multithreading

For this processor, we can look where things are shared between threads and other part are not shared. There is only **5%** more area of circuit added to be able to have multithreading. This is a very beautiful idea right? It costs nothing and makes us gain a lot.

Now we have done everything that is currenlty in our processor, we are done. Now in your phone, laptop, etc... you have super scalar Multithreading processor.

4.6.3 Non Blocking Caches

Let us consider the next example:

```
lw $t2 0($t0) # t2 = mem[t0]
lw $t3 0($t1) # t3 = mem[t1]
addi $t3, $t3, 123
andi $t3, $t3, 0xff
```

If there is a cache mis for `mem[t0]`, one need to **wait** for the (slow) main memory. At the moment, our cache works as a finite state machine, we ask him some element, it goes search in his cache, if it has it → gives it to us. If not, then it has to look for it in the main memory. But this is not a pipeline, this is a **finite state machine**. This means: we have to **wait until** it is done, then we can continue our program.

But us, as superscalar processor, we would want to **continue execution** as far as dependencies permit it.
But don't we have a solution when we want to have parallelism?

pipeline

Nonblocking Caches

The cache controller could save a request, while waiting for the main memory, if the data are in the cache (**hit under miss**)

- Hide the miss latency with useful work

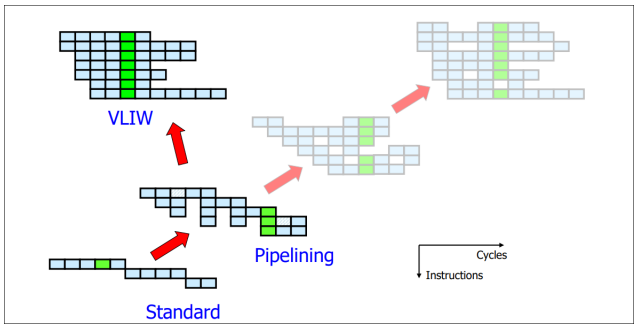
The cache controller could save a request, while waiting for the main memory, by issuing another request to memory (**miss under miss**)

- Overlap the latency of the two misses

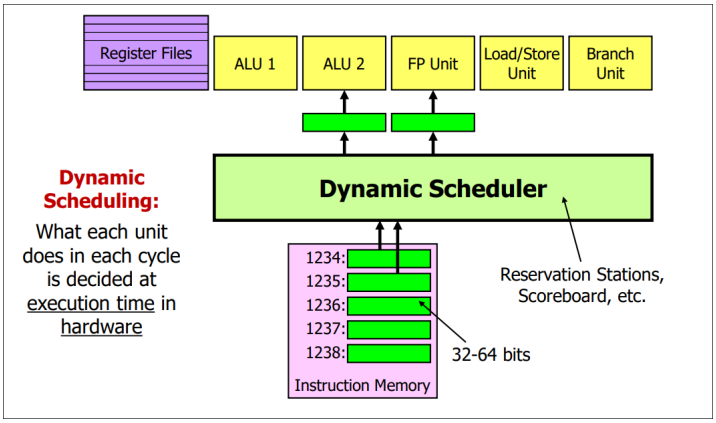
Nonblocking caches are generally needed for dynamically scheduled superscalar processors

4.6.4 Very Long Instruction Word (VLIW) Processor

The question we will ask now is: Is there a whole completely different way of extracting instruction level parallelism? A fundammentaly different way of doing it:

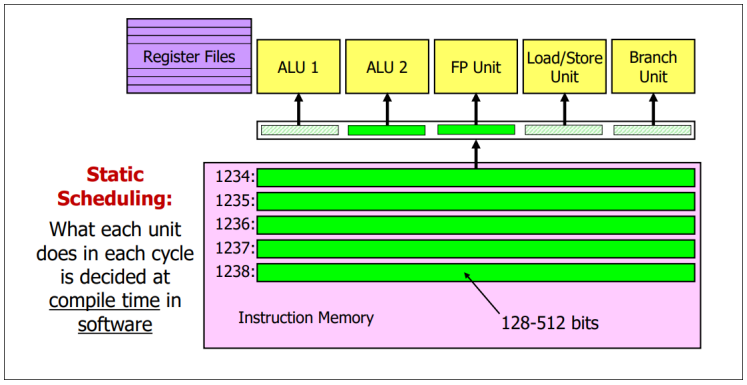


First let us take dynamically scheduled superscalar processor (kind of):



This thing that we build is very fast. When we say fast we are talking 0.2 nanosecond. The decision that we are taking here is pretty hard, are all the argument ready, do we have the tag ready etc...in only 0.2 nanosecond. But the dependency that we are making a decision of was known from the beginning. From the start of the program we **knew** that there will be a data dependency here. But we choosed to do it at the last minute. But the dynamic scheduler takes a lot of space in our circuit. We could have as many transistors in all execution units as in the dynamic scheduler.

At this day the limiter is the dynamic scheduler. What don't we turn the dynamic into static:



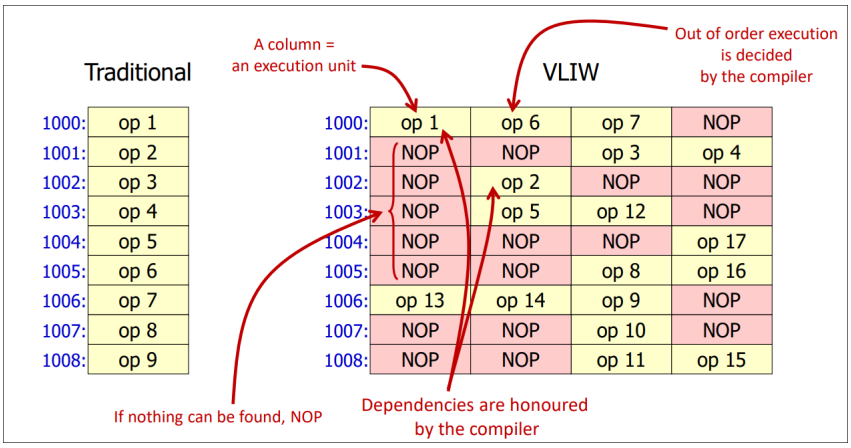
Let us take a decision **a priori** before we start executing about what we execute, at each cycle.

Definition 9 (Static Scheduling)**.** What each unit does in a cycle is decided at compile time in software

So now instead of having small instruction that use one exeuction unit at a time, let us use big instruction that use all the execution units at the same time. **What are the problems?**

For this, we need to know a lot of information about the architecture (as a software person). Before we didn't need to know because all the issue were solved in hardware (this is why we used to resolve issues at the last minute)
Secondly remember that every year our transistors becomes better and cheaper, the way we used to speed up our processor was to add more unit, a second alu, a second fp unit, etc. But know as we now use **static scheduling** we cannot change our execution units at of nowhere. So when buying a new processor, if we run a program that use the same set of instruction as the previous processor, there won't be any difference of speed.

But now what would look like a traditional code into a VLIW code:



The left one means 'every things we need to do'. The left code means 'every thing that we need to do at exaclty each cycle'. This is a great idea right? We don't have an issue about the size of our program, at the current time, laptop usually 16-32 GiB of ram, a usual program takes aroung 1MiB, we are three order of magnitudes away, so this is fine. Our program after being used in the ram it is also put in the cache, in the **l1 cache**. But this cache **care about size** it needs to, it is very small and polluting it with `nop` is very bad. **Challenges of VLIW**

- **Compiler Technoloy**: Most sever limitation until the end of the 90s (VLIW idea is aroung since the 70s)
- **Code Bloating**: all the `nop` occupy memory space and thus **cost**
- **Binary Incompatibility**

Compiler Technology First let us check what kind of information is missing at compile time? Let us take a program and scheduled it:

```
loop: ls $f0, 0($r1)
          add $f4, $f0, $f2
          sd ($srl), $f4

          subi $r1, $r1, 8
          bnez $r1, loop
```

- Schedule on a VLIW processor
  - Slot 1: Load/Store Unit or Branch Unit
  - Slot 2: ALU
  - Slot 3: Floating-Point Unit
- Latencies
  - Load/Store → 2 cycles
  - Integer → 2 cycles
  - Branch → 2 cycles
  - Floating-Point → 3 cycles

So let us build the table for it:
This first instruction is kind of easy to construct, we know that there won't be any issue with it so let us just put it in there

Load/Store/Branch Unit	ALU	Floating-Point Unit	
ld			Cycle 1
			Cycle 2
		ADD	Cycle 3
			Cycle 4
	subi		Cycle 5
sd			Cycle 6
bnez			Cycle 7
			Cycle 8
			Cycle 9

To build the scheduling table, we proceed cycle by cycle, respecting both the **structural constraints** (one instruction per functional unit per cycle) and the **data hazards**, while accounting for the **fixed instruction latencies** provided by the VLIW architecture.

Step 1: Identify Functional Units and Constraints

The VLIW processor provides three execution slots per cycle:

- **Slot 1**: Load/Store *or* Branch Unit
- **Slot 2**: Integer ALU
- **Slot 3**: Floating-Point Unit

Only one instruction can be issued per slot per cycle, and instructions must respect producer-consumer dependencies.

Step 2: Instruction Latencies

We must delay the consumer of a value until the producer has completed:

Instruction Type	Latency
Load/Store	2 cycles
Integer ALU	2 cycles
Branch	2 cycles
Floating Point	3 cycles

Step 3: Schedule the Load Instruction

```
ls $f0, 0($r1)
```

- This instruction uses the **Load/Store unit**.
- It produces `$f0`, which is needed by the floating-point `add`.
- Since the load latency is **2 cycles**, `$f0` becomes available at the **end of Cycle 2**.

Thus, we place the load in **Cycle 1**, Slot 1.

Load/Store	ALU	FP	Cycle
ld			1
			2
		ADD	3
			4
	subi		5
sd			6
bnez			7
			8
			9

Step 4: Handle the Floating-Point Dependency

```
add $f4, $f0, $f2
```

- This instruction depends on `$f0`.
- `$f0` is available only after **Cycle 2**.
- Therefore, the earliest possible issue is **Cycle 3**.
- It uses the **Floating-Point unit** and has a latency of **3 cycles**.

We place it in **Cycle 3**, Slot 3.
Cycles 2 and 4 are idle in the FP unit because:

- Cycle 2: data not ready

- Cycle 4: FP instruction still executing (latency)

Step 5: Schedule the Integer Instruction

```
subi $r1, $r1, 8
```

- Uses the **ALU**
- Independent of the floating-point operation
- Can be scheduled as soon as the ALU is free

We place it in **Cycle 5**, Slot 2.

Step 6: Schedule the Store Instruction

```
sd ($srl), $f4
```

- Depends on `$f4`, produced by the FP `add`
- FP latency is **3 cycles**, starting at Cycle 3
- `$f4` becomes available at the **end of Cycle 5**

Thus, the store can be issued in **Cycle 6**, Slot 1.

Step 7: Schedule the Branch Instruction

```
bnez $r1, loop
```

- Depends on `$r1`, updated by `subi`
- `subi` latency is **2 cycles**, starting at Cycle 5
- `$r1` becomes available at the **end of Cycle 6**

We place the branch in **Cycle 7**, Slot 1.

Step 8: Branch Latency and Empty Cycles

- The branch has a **2-cycle latency**.
- The processor cannot know the next fetch address until the branch resolves.
- This results in **Cycles 8 and 9** being empty.

These empty cycles highlight an important limitation of **static scheduling**: the compiler must conservatively assume worst-case behavior because it lacks **runtime information**, such as actual branch outcomes or memory access delays.

Key Takeaway

This table illustrates what is **missing at compile time**:

- Exact branch behavior
- Memory access variability
- Dynamic instruction overlap opportunities

The question we have is: are we just unlucky with our program, is it just a horrible program and that; normally it is way better than this one? **Which one is better?**

A natural question to ask is whether this poor schedule is simply the result of bad luck, or whether static scheduling is inherently limited. To answer this, we can compare the behavior of our VLIW processor with that of a **dynamically scheduled processor**.
With dynamic scheduling, the processor can make decisions at runtime using information that is not available to the compiler. First, registers can be **renamed dynamically**, which helps eliminate false dependencies and allows independent instructions to execute earlier. Second, and more importantly for our example, the processor can use **branch prediction**. If the branch predictor correctly predicts that the loop continues, the processor can speculatively fetch and execute instructions from the next iteration. In this case, the two-cycle branch latency can be effectively hidden, meaning that after the first iteration we can gain up to two cycles per loop iteration "for free."
Furthermore, dynamic scheduling allows the processor to react to variations in memory access latency and execution timing. When a load completes earlier than expected, dependent instructions may be issued immediately, rather than waiting for a conservatively assumed latency as in static scheduling. This flexibility generally leads to higher utilization of functional units and fewer idle cycles.
However, this improvement does not come without cost. Dynamically scheduled processors require more complex hardware, including reorder buffers, reservation stations, and branch predictors, which increases power consumption and design complexity. In contrast, VLIW processors shift this complexity to the compiler, resulting in simpler hardware but potentially less efficient execution when compile-time assumptions are overly conservative.
In summary, dynamic scheduling tends to perform better for irregular code and control-heavy loops, such as the one considered here, while static scheduling can be effective when program behavior is highly predictable and well understood at compile time.

Remark 50. At the computation level, there is not big difference in terms on how we executed the code between the dynamic scheduling and the static scheduling. In both cases we will have the same 'parallelism', the difference is that now, the compiler does the job of the dynamically scheduler.

The issue for us at the moment is that in a static scheduler, we have to wait for each iteration: we cannot do the `ld $f0, ($r1)` before having finished the `bnez $r1, loop` instruction. On the other hand, when we used a dynamically scheduled processors we were able to do so (using branch prediction). Is there a way for the static scheduler to do the same job (execute instruction that is in the next iterations of the loop)?

Enlarge the Scop for ILP: Loop Unrolling

Yes we can! The goal for us would be: instead of seeing it as a big loop; why not just **unroll** the loop:

```
Loop: ld $f0, ($r1)
      addd $f4, $f0, $f2
      sd ($r1), $f4
      subi $r1, $r1, 8
      bnez $r1, Loop
```

$\Longrightarrow$

```
Loop: ld $f0, ($r1)
      addd $f4, $f0, $f2
      sd ($r1), $f4
      ld $f6, ($r1-8)
      addd $f8, $f6, $f2
      sd ($r1-8), $f8
      ld $f10, ($r1-16)
      addd $f12, $f10, $f2
      sd ($r1-16), $f12
      ld $f14, ($r1-24)
      addd $f16, $f14, $f2
      sd ($r1-24), $f16
      ld $f18, ($r1-32)
      addd $f20, $f18, $f2
      sd ($r1-32), $f20
      subi $r1, $r1, 40
      bnez $r1, Loop
```

But this is nice for us; now we can also do renaming of registers as we used to do. We have 5 different copy of the same code this allows us to rename our register. So here if we take the second load for instance; it depends only on `$r1` so we can already execute it right after the first load → we can do the same for all loading and storing. For the add, each add depends on each load and is shifted of two cycles. So now the output would be something like this:

Cycle	Load/Store/Branch Unit	ALU	Floating-Point Unit
1	`ld $f0, ($r1)`	`nop`	`nop`
2	`ld $f6, ($r1-8)`	`nop`	`nop`
3	`ld $f10, ($r1-16)`	`nop`	`addd $f4, $f0, $f2`
4	`ld $f14, ($r1-24)`	`nop`	`addd $f8, $f6, $f2`
5	`ld $f18, ($r1-32)`	`nop`	`addd $f12, $f10, $f2`
6	`sd ($r1), $f4`	`nop`	`addd $f16, $f14, $f2`
7	`sd ($r1-8), $f8`	`nop`	`addd $f20, $f18, $f2`
8	`sd ($r1-16), $f12`	`nop`	`nop`
9	`sd ($r1-24), $f16`	`nop`	`nop`
10	`sd ($r1-32), $f20`	`subi $r1, $r1, 40`	`nop`
11	`nop`	`nop`	`nop`
12	`bnez $r1, Loop`	`nop`	`nop`
13	`nop`	`nop`	`nop`

Table 4.1: Instruction scheduling across functional units

Now we have **26** cycles (vs. 90 cycles from before)

What kind of information is missing at compile time

For example let us consider this code:

```
sw x3, 456(x1)
lw x2, 123(x4)
```

We want to ask wether or not there is a RAW dependence?

- At run time:
  - Check if `x1 + 456 = x4 + 123`
  - Forwarding may even hide the memory latency...
- At compile time:
  - ??? (special techniques: alias analysis)

But is there a big issue with loading? What happens when we are loading but we have a cache miss? We will have to wait for like 100 cycles but we were trying to do parallelism? Now we have to wait 100 cycles for us to have the value we are looking for? This is horrible, how can we resolve that? Should we just assue that we will need to wait 100 cycles everytime we have a load instruction? If so then the cache becomes just useless.
Can we do better? I am optimist maybe I can just say 'my cache is good' and hope for a cache hit; after one cycle I use the value that the load gave us.
So now we are running our code as we get a cache hit for a couple of cycle but then the cache sends us a signal 'cache miss' → we stall everything. The difference about this stall and how we used to do this. Now even independent instruction **has to wait for us**. The issue is that we have the alu unit, load/store unit, …that are all together in one instruction, so if one of them needs to wait, everybody has to wait with him.

VLIW Compilation Techniques

There many old and new techniques:

- Aliasing analysis
- Loop unrolling, peeling, fusion, and distribution
- Software pipelining, modulo scheduling
- Trace scheduling, superblock scheduling
- With hardware support in the processor: Predication, hyperblock, scheduling, ...

usually advantage **not for free**:

- Faster only on most frequent part of the code; penalty elsewhere
- Difficulties to apply them in the general case
- Larger code (worsens the performance of the l-cache)

4.6.5 Summary of chapter 4

So which architecture should we use between VLIW and Superscalar?
In most commont, general purpose computing machine, we use dynamically scheduled, superscalar, processor with speculative execution, SMT. This means that in our smartphone, laptop, pc, etc. We use superscalar processor. On the other hand the VLIW architecture is used in some **embedded applications**(e.g., DSP), everything such as voice processor, filter, etc... And the reason one this is better for embedded application and/or signal processing machine, is also the is way less costly as the other superscalar, we don't have to have all those unit and dynamic scheduler which is useless for that type of work. Furthemore maybe we don't even need a cache in here in the first place.

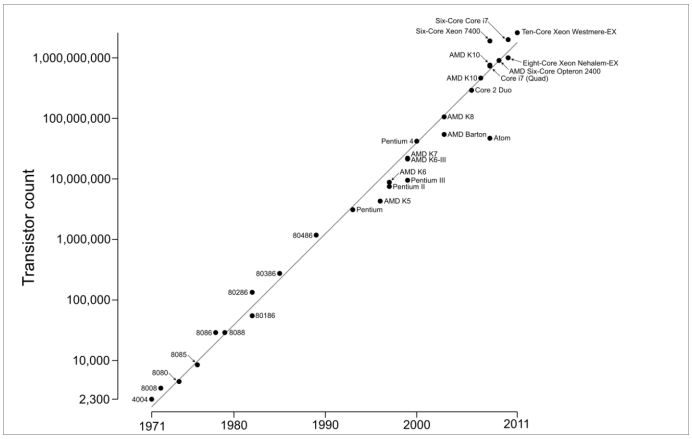
- Dynamically-scheduled superscalar processor are the commercial state-of-the-art in general purpose computing (laptops, data centers): current high-end implementations of **x86 (Intel and AMD)** as well as **ARM** are all superscalar
- VLIW/EPIC processors represent an alternative, valuable in some situations: **Itanium 2** was a failed attempt by Intel to bring VLIWs in general-purpose computing; yet, practically all digital signal processors are VLIW (e.g., in all smartphone or for audio processing)
- Performance is the result of a **subtle balance** between exploiting possibilities in compilers aNd managing hardware implementation difficulties

4.7 Intel x86 and ARM

Processor	Date	f (MHz)	Trans.	Features
4004	4/71	0.108	2.3k	First μP
8008	4/72	0.108	3.5k	First 8-bit μP
8080	4/74	2	6k	Popular 8-bit
8086	6/78	5–10	29k	First 16-bit μP; 20-bit addressing
8088	6/79	5–8	29k	Simpler; IBM PC
80286	2/82	8–12	134k	Protected mode; 24-bit addressing
80386	10/85	16–33	275k	32-bit (x86)
80486	4/89	25–100	1.2M	Pipelined (5-stage); cache
Pentium	3/93	60–233	3.1M	Superscalar; dual pipeline
Pentium Pro	3/95	150–200	5.5M	Out-of-order; L2 cache
Pentium II	5/97	233–400	7.5M	MMX (SIMD instructions)
Pentium III	3/99	450–1400	9.5–28M	SSE (incl. SIMD-FP); 10-stage pipeline
Pentium 4	12/00	1300–2200	42M	SSE2 (128-bit); TC; 20-stage pipeline
NetBurst	04/04	800–3800	376M	64-bit (x86-64); SSE3; HT; 31-stage pipeline
Core	06/06	up to 3300	105M	SSSE3; SSE4; 12–14-stage pipeline
Nehalem	11/08	up to 3600	731M	20–24-stage pipeline
Sandy Bridge	01/11	up to 4000	–	AVX (256-bit, 3-op); 14–19-stage pipeline
Haswell	06/13	up to 4400	–	AVX2
Skylake	08/15	up to 3700	–	AVX-512 (512-bit)

Table 4.2: Evolution of Intel Microprocessors

Interl Processors As we have seen before the growth of the number of transistors for processors is exponential:



Remember that we want to stay at the same price for our processors → we adde more transistors in our processors this is the reason for the growth of transistors. But this number of transistors may not be that useful.

So here the Intel Itanium 2 9050 and Intel Itanium 9150M are the only LVIW processors here. As we can see in the Out of Order line there is *none* in both of them.

Remark 51. As we can see there is 1.72 **billion** transistors for those two processors. But didn't we say that we needed less transistors for them? Yes in theory, but only if you are focusing on one type of program. This cannot work for general purpose program.

Processor	Intel 1-core Xeon	AMD 1-core Opteron 854	Intel 2-core Xeon X5270	AMD 2-core Opteron 8224SE	Intel 4-core Xeon X7350	AMD 4-core Opteron 8360SE	Intel 6-core Xeon X7460
Bit-width	32/64	32/64	32/64	32/64	32/64	32/64	32/64
Cores/chip × threads/core	1 × 2	1 × 1	2 × 1	2 × 1	4 × 1	4 × 1	6 × 1
Clock rate	3.80 GHz	2.80 GHz	3.50 GHz	3.20 GHz	2.93 GHz	2.50 GHz	2.67 GHz
Cache L1-L2-L3	12K/16K – 2M	64K/64K – 1M	2 × 32K/32K – 6M	2 × 64K/64K – 2 × 1M	4 × 32K/32K – 2 × 4M	4 × 64K/64K – 4 × 512K	6 × 32K/32K – 3 × 3M
Execution rate/core	3 instr.	3 instr.	1 complex + 3 simple	3 instr.	1 complex + 3 simple	3 instr.	1 complex + 3 simple
Pipeline stages	31	12 int / 17 fp	14	12 int / 17 fp	14	12 int / 17 fp	14
Out-of-order	126	72	96	72	96	72	96
Memory bus	800 MHz	6.4 GB/s	1333 MHz	10.6 GB/s	1066 MHz	10.6 GB/s	1064 MHz
Package	LGA-775	uPGA 940	LGA-771	LGA-1207	LGA-771	LGA-1207	LGA-771
IC process	90 nm	90 nm	45 nm	90 nm	65 nm	65 nm	45 nm
Die size	109 mm^2	106 mm^2	107 mm^2	227 mm^2	2 × 143 mm^2	283 mm^2	503 mm^2
Transistors	169 M	120 M	410 M	233 M	2 × 291 M	463 M	1900 M
List price	$903	$1,514	$1,172	$2,149	$2,301	$2,149	$2,729
Availability	3Q05	3Q05	3Q08	3Q07	3Q07	2Q08	4Q08
Scalability	1–2 chips	2–4 chips	1–2 chips	1–4 chips	1–4 chips	2–4 chips	1–4 chips
SPECfp2006	11.4 / 11.7	11.2 / 12.1	26.5 / 25.5	14.1 / 14.2	21.7 / 18.9	14.4 / 18.5	22.0 / 22.3
SPECfp2006_rate	20.9 / 18.8	41.4 / 45.6	84.9 / 57.7	105 / 96.7	184 / 108	170 / 156	274 / 142
x86 codename	Irwindale	Athens	Wolfdale	Santa Rosa	Tigerton	Barcelona	Dunnington
Microarchitecture	NetBurst	K8	Core	K8	Core	K10	Core

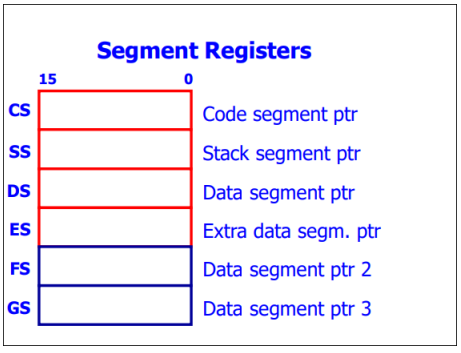
Processor	Intel Itanium 2 9050	Intel Itanium 2 9150M	IBM POWER5+	IBM POWER6	Fujitsu SPARC64 VI	Fujitsu SPARC64 VII	Sun UltraSPARC T2+
Bit-width	64	64	64	64	64	64	64
Cores/chip × threads/core	2 × 2	2 × 2	2 × 2	2 × 2	2 × 2	4 × 2	8 × 8
Clock rate	1.60 GHz	1.67 GHz	2.20 GHz	5.00 GHz	2.40 GHz	2.52 GHz	1.40 GHz
Cache L1-L2-L3	2 × 16K/16K – 12M(on)	2 × 16K/16K – 12M(on)	2 × 64K/32K – 1.92M(off)	2 × 64K/64K – 32M(off)	2 × 128K/128K – 6M	4 × 64K/64K – 6M	8 × 8K/16K – 4M
Execution rate/core	6 issue	6 issue	5 issue	7 issue	4 issue	4 issue	16 issue
Pipeline stages	8	8	15	13	15	15	8 int / 12 fp
Out-of-order	None	None	200	Limited	64	64	None
Memory bus	8.5 GB/s	10.6 GB/s	12.8 GB/s	75 GB/s	8 GB/s	8 GB/s	42.7 GB/s
Package	mPGA-700	mPGA-700	MCM-5370 pins	N/A	412 I/O pins	412 I/O pins	1831 pins
IC process	90 nm	90 nm	90 nm	65 nm	90 nm	65 nm	65 nm
Die size	596 mm^2	596 mm^2	245 mm^2	341 mm^2	421 mm^2	400 mm^2	342 mm^2
Transistors	1.72 B	1.72 B	276 M	790 M	540 M	600 M	503 M
List price	$3,692	$3,692	N/A	N/A	N/A	N/A	N/A
Availability	3Q06	4Q07	4Q05	2Q08	2Q07	3Q08	2Q08
Scalability	1–64 chips	8–128 chips	1–32 chips	2–32 chips	4–64 chips	4–64 chips	2 chips
SPECfp2006	14.5 / 17.3	N/A	10.5 / 12.9	15.8 / 20.1	9.7 / 21.7	10.5 / 25.0	N/A
SPECfp2006_rate	1534 / 1671	2893 / N/A	197 / 229	1837 / 1822	1111 / 1160	2088 / 1861	142 / 111
Architecture status	Inactive	Active	Inactive	Active	Inactive	Active	Active

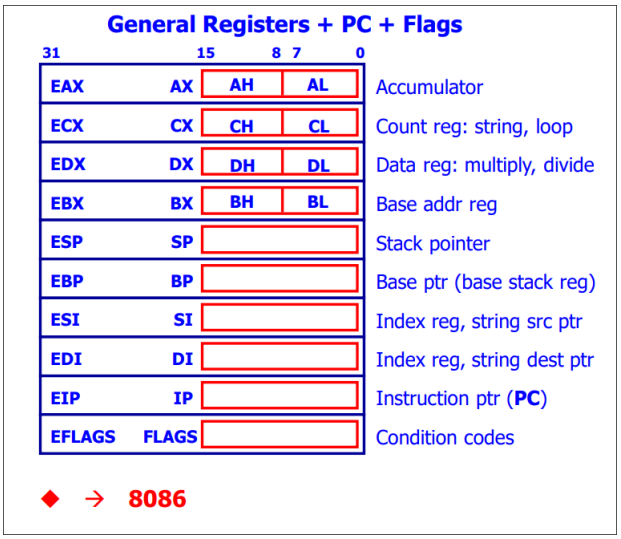
Legacy x86 Features

- Very **small number of registers**, partly dedicated or **specialised**
- Natively 16-bit, extended to 32 in successive steps requiring backward compatibility (e.g., 3 modes for address generation)
- **Highly variable instruction length** and encoding (1 to 17 bytes in original x86, prefixes, postfixes, etc.)
- **CISC** instruction set

Registers

There is a very small number of general purpose registers (approx. 4 integer plus 8 FP – not shown, versus 32+32 typ. RISC)

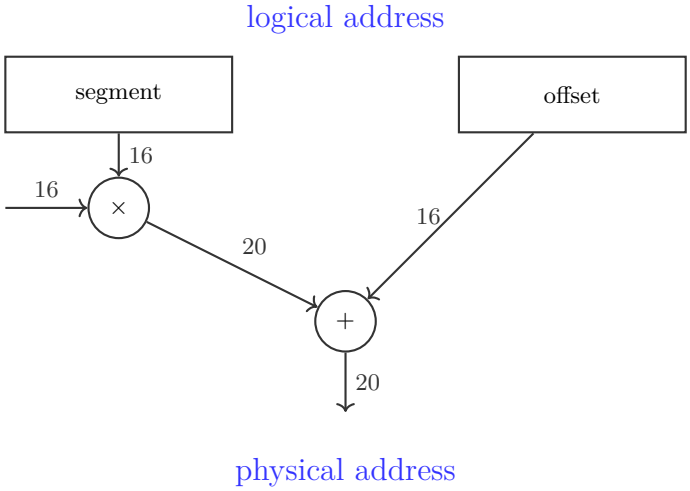




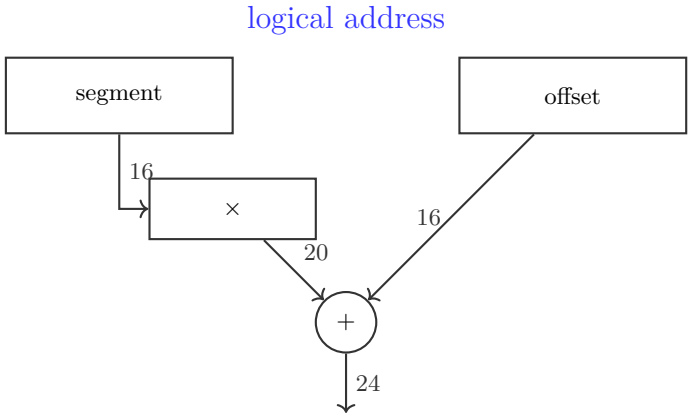
- small number of registers makes spilling more frequent
- Advanced compiler technique (e.g., loop unrolling) increase registers pressure
- Partial specialization of the registers makes **effective compiler use difficult**

Memory Address

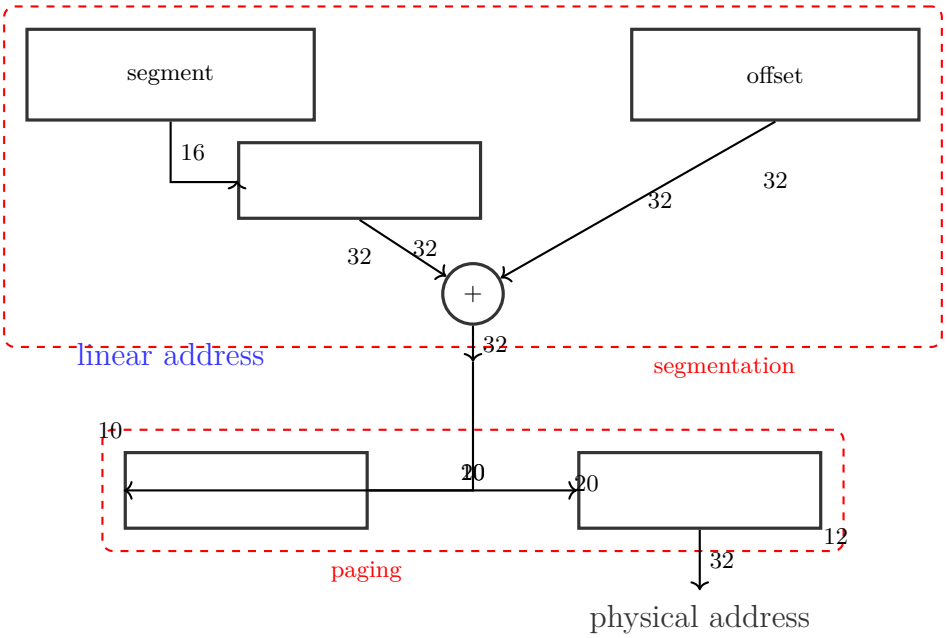
You can also see the history of the memory. First we need a physical address that is more than 16 bits (our memory is greater that 2^{16}) However our architecture at that time is only at 16 bits. We need another register in order for have access to the address.



This is the real mode (8086)
Now if we take a look at the protected mode (80286), now we have a sort of multiprogramming.



Now if we go even later in the protected mode (80386, 80486, Plentium). we need now to have a bit of multiprogramming right?



Ce schema est faux si j'ai la force de change un peu

Operand Types

- **Not** a load/store architecture

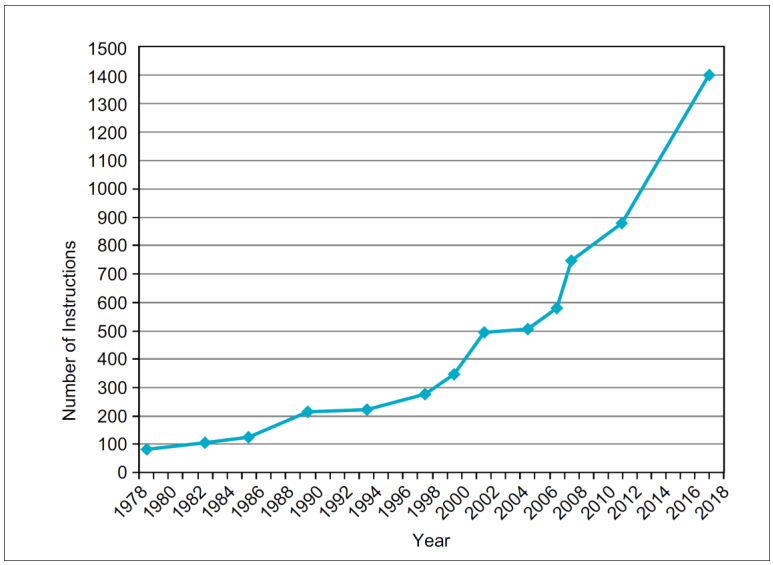
Source 1 = Destination	Source 2
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

Addressing Mode

- **Indexed with displacement** → [base + reg + reg + displacement]
  - Same registers as in mode indexed
- **Scaled indexed** → $\left[\text{base reg} + 2^{\text{scale}}x \text{ reg}\right]$
  - Only in 32 bit mode
  - Scale is 0, 2, or 3
  - Index register can be any of the basic registers (except ESP)
  - Base register can be any of the basic registers
- **Scaled indexed with displacement** →

$$\left[\text{base reg } + 2^{\text{scale}}x \text{ reg } + \text{displacement}\right]$$

Number of x86 instructions over time



Remark 52. This part is not finished

Chapter 5

Multiprocessors

As the end of the course approaches, the focus remains on continuing the same fundamental ideas, but through different mechanisms. Much of modern computer architecture relies on the continuous improvement of transistors: they become smaller, faster, and cheaper, allowing architects to use more of them to achieve greater functionality and performance.

These transistor resources have been invested primarily in two directions. First, they have been used to create features that benefit programmers by abstracting hardware complexity. Examples include virtual memory, which hides details such as physical memory size, location, and addressing, thereby simplifying software development and reducing programmer burden.

Second, transistors have been heavily invested in improving system performance. A major example is the use of caches, which consume a large number of transistors but help mitigate the growing performance gap between processors and memory. Since memory technologies do not scale as well as logic, additional transistors are used to reduce memory access latency and improve overall speed.

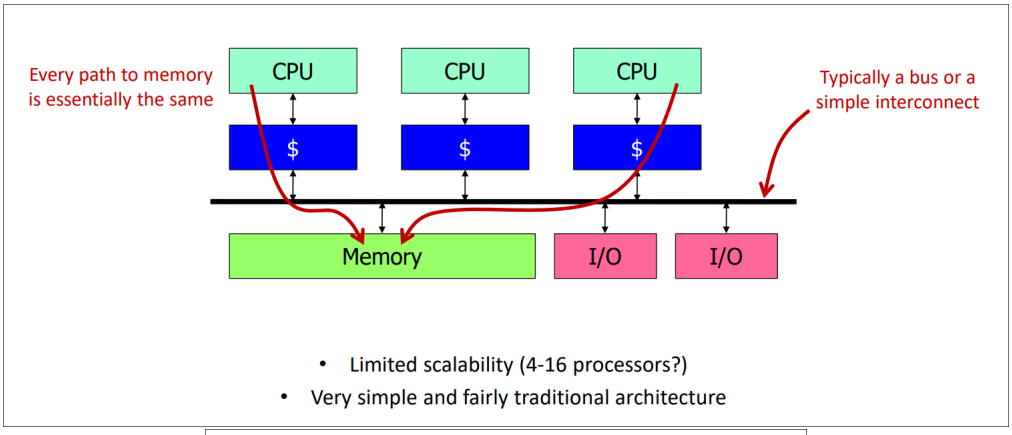
More recently, transistors have been devoted to designing processors that execute programs faster through dynamically scheduled, out-of-order superscalar architectures. These processors improve performance in two main ways. One approach is through complex control and management logic that extracts instruction-level parallelism from fundamentally sequential programs. This requires hardware to detect and manage data dependencies, which is transistor-intensive.

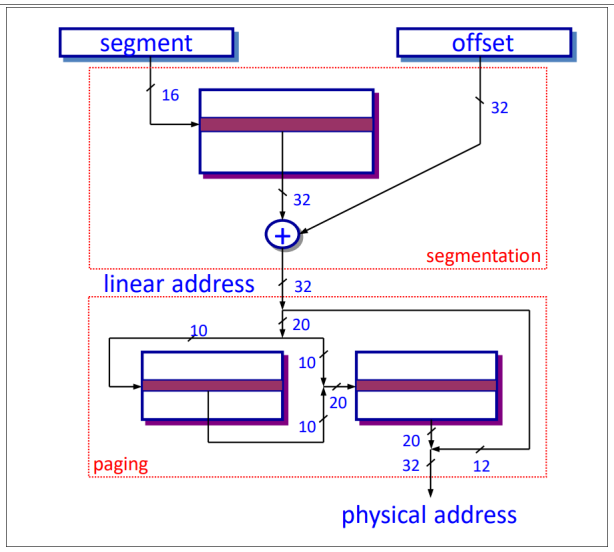
The second approach is to add multiple execution units, such as several floating-point units, to allow more instructions to be executed simultaneously. However, this strategy is fundamentally limited by data dependencies: when an instruction depends on the result of another, execution must wait. While processors can look ahead in the instruction stream to find independent instructions, the available parallelism is ultimately constrained by these dependencies.

This chapter will try answer the question: where should we put our transistors when we have too much of them? → adding processors.

5.1 Cache Coherence

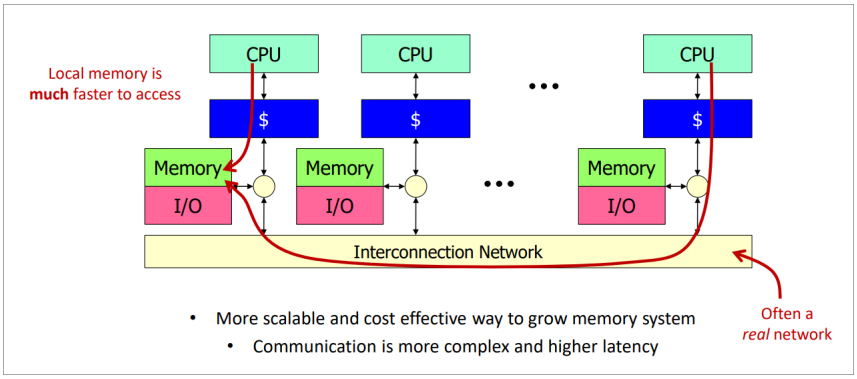
Flynn's Taxonomy (1966)
How does it works is: we take our normal system (memory bus, i/o, etc.) and **on the same bus**, we add all of our CPU. **Share-Memory Multiprocessors**





Distributed-Memory Multiprocessors

The other way of doing it is to take all of our cpus and construct little system around each of them. And then link them all together using the Interconnection Network. This way we have more scalability as the previous multiprocessors.



Remark 53. What we mean here by 'Often a real network' is that all those system doesn't have to be physicallly connected. For instance https://setiathome.berkeley.edu/ was a famous distributed computing project where people volunteered their computer's idle processing power to

help analyse radio signals from space in the search of extraterrestrial intelligence. Instead of having like a very high tech optic fiber Interconnection Network, all the computer were connected with the slow internet.

Programming Paradigms

Shared-Memory

- Data exchanged **implicitly** through shared variables in a comon memory space
- Standard libraries (e.g., **OpenMP**) simplify programming
- Natural on shared-memory architectures (e.g., **SMP**, **NUMA**)
- Can be implemented as **Distributed Shared Memory** (**DSM**) on systems with physicallly distributed memory, leveraging virtual memory abstraction (e.g., **TreadMarks** for DSM; **Apache Spark** (scala!!) for a DSM-like abstraction in big data)

Message Passing

- Data exchanged **explicitly** by sending and receiving messages over a network or interconnect
- Standard libraries (e.g., **MPI**) are widely used
- Natural on distributed-memory systems with private memory per processor
- Can also be implemented on shared-memory systems (e.g., **NUMA**), though it may introduce unnecessary overhead compared to native shared-memory programming

Accessing Memory in Distributed Systems

Suppose a CPU wants to access memory that physically belongs to another CPU. There are two main ways to achieve this: a **hardware-based approach** and a **software-based approach**.

Hardware-based approach (NUMA / Hardware DSM)

In a hardware solution, the system exposes a single shared address space, even though memory is physically distributed across processors. Each cache is connected to an interconnection network, often through a dedicated controller (as the yellow node between cache and memory).

When the cache performs a lookup:

- If the access is a **cache hit**, execution proceeds normally.
- If the access is a **cache miss**, the cache controller examines the address.

Modern systems have large address spaces (e.g., 64-bit), allowing some bits of the address to encode **which processor owns the data**.

- If the address corresponds to **local memory**, the cache fetches the data normally.
- If the address corresponds to **remote memory**, the cache controller generates a small request packet and sends it over the interconnection network to the owning processor.

The remote processor then:

- Accesses its local memory,
- Sends the requested data back over the network,
- Allows the requesting cache to be reloaded.

Because cache refills are on the critical execution path, the interconnection network must be **very fast**. Although remote accesses are already costly (e.g., hundreds of cycles), they must not imply millisecond-scale delays.

This approach effectively implements a **hardware Distributed Shared Memory (DSM)**. The memory is:

- **Distributed** physically,
- **Shared** in terms of addressing,
- Managed directly by hardware.

Such systems are classified as **NUMA** architectures, since access latency depends on whether the memory is local or remote.

software-based approach (software DSM)

The same shared-memory abstraction can also be implemented in software. Consider two independent PCs connected by a network: each has its own memory, operating system, and private address space.

At first glance, sharing memory seems impossible. However, modern systems already rely heavily on **virtual memory**. Every memory access goes through the **TLB**, which is under the control of the operating.

The operating system can exploit this as follows:

- If a requested page is not present locally, the OS detects this via a TLB miss.
- Traditionally, the missing page is fetched from disk.
- In a software DSM system, the missing page may instead reside in the memory of another machine.

From the program's perspective, the memory still appears shared. The OS transparently:

- Fetches pages over the network,
- Updates page tables and the TLB,
- Moves pages between machines as needed.

When multiple processors frequently access the same data, pages may migrate back and forth between machines, effectively implementing communication through shared memory. This behavior is sometimes described as **page ping-pong**.

Although slower than hardware DSM, software DSM can outperform disk access and provides a powerful abstraction by reusing mechanisms already present in virtual memory systems.

Why (Hardware) Shared Memory

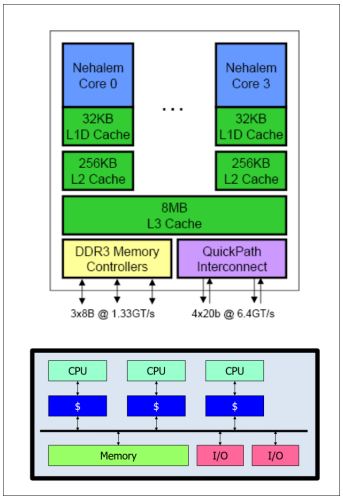
Advantages	• For applications looks like a multitasking uniprocessor • For OS only evolutionary extensions required • Easy to do communication without OS • Software can worry about correctness first, then performance
Disadvantages	• communication is implicit, hence harder to optimize • Propoer synchronization is complex • Hardware designers must implement
Result	• **Symmetric Multiprocessors** (SMPs) where the foundation of early supercomputers but gave way to distributed-memory

message-passing systems as scaling issues → made SMPs less efficient

- **Chip Multiprocessors (CMPs)** or **multicore** processors dominate as the most widespread form of parallel computing, driving multibillion-dollar market

Intel Nehalem (2008)

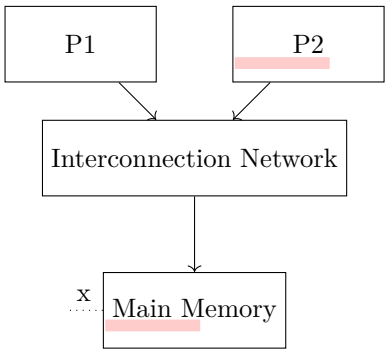
If we take an 'old' architecture, we can see that it is done like we have seen before:



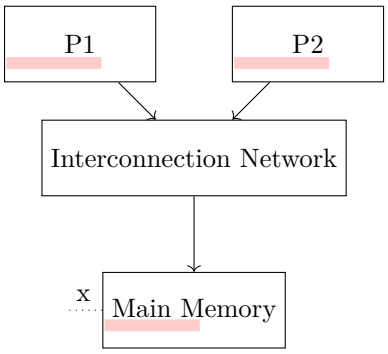
Cache Coherence Problem Step 1

Imagine we have this sequence of processors that have their own private caches. When the second processor load a particular value in memory (a miss) is brough up to the processor and now is in the cache.

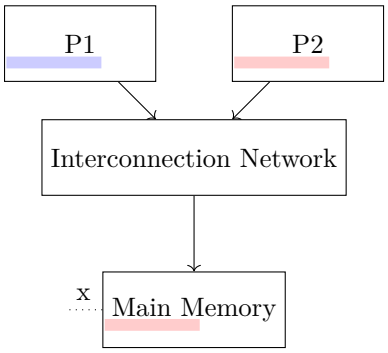
`ld r2, x`



Suppose now that the processor 1 does the same `ld r2, x`, we would get without any problem:



Now both processors have the value of the address `x` in the cache. Now suppose that processors one want the change this value in memory, this new value will be only overwritten in the cache:



Then when the second processor will try to read `ld r5, x` we will have an error, the `r5` register will have the previous value (the pink) and not the correct one (blue).
And this is not a matter of the cache not being **write-through**
But this issue is not really new in multiprocessors, if we took an i/o like a button for instance. We poll the button one time, we store the result in cache. When we want to repolling the button → cache hit → not pressed. But maybe it is pressed.
We can go even with the DMA, the DMA works kind of the same way as the processor one in our case. What can we do?
But there we have simpler ad-hoc ways to handle this:

- Flush the cache in software
- Invalidate caches lines in software
- Define **noncachable areas of memory** and perform I/O there

Remark 54. The issue that we have here is not an issue of time, the p2 memory will **never know** the blue data. The only way for the p2 cache to have the blue is:

- blue has to be in main memory: p1 access another data in memory which is in the same line as the blue one (in the cache) which make the cache **evict** the blue data → needs to be written in memory
- It has to **throw away** the pink data on the p2 cache (**and not write it in main memory**)
- reload the data in p2 from main memory

Coherent Memory System

1. **Preservation of program order**. If P writes in X and then P reads in X, and in the meantime no other processor has written X, the value returned is the value previoulsy written by P
2. **Coherent view** if P1 write X and P2 read X, and in the meantime no other processor has written X, the value returned is the value previously written by P1, if the read and write are sufficiently separated in time.
3. **Write Serialization** if P1 writes X and P2 writes X, all processors see the writes in the same order

Snoopy Cache-Coherence Protocols

So what we will need to do is actually have our p2 processor/cache **looking** to main memory to see if any changed is required. The way of doing this is call **snoopy cache-coherence protocols** which is the most intuitive and practical method of resolving this issue. We are snooping '*mettre son nez*' *Paulo Ienne*:

Bus provides serialization point

- The bus is used the serialize operations, meaning it ensures that all caches see memory updates in consistent order. If one processor writes a value to memory, the bus shouts this action to all caches, so they can all 'mettre leur nez' and update their copies of the data if needed.

Each cache controller snoops all bus transactions

Each cache controller listens to the bus for transactions that involve memory addresses **it holds**. When it 'snoops' a transaction, it determines wether it needs to:

- Invalidate
- Update
- Supply value

Which one depends on state of the line and the protocol. State? does that means that we have a fsm??

FSM of a Cache

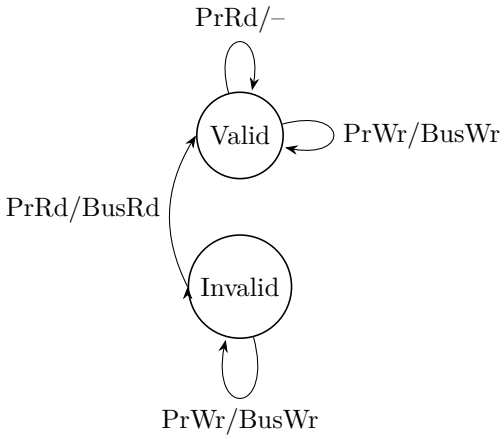
Imagine that we want to describe a cache in a formal way (forget what we have seen before). For our caches we have:

- write-through, write-no-allocate cache
- action
  - PrRd (processor read)
  - PrWr (processor write)
  - BusRd (bus read)
  - BusWr (bus write)

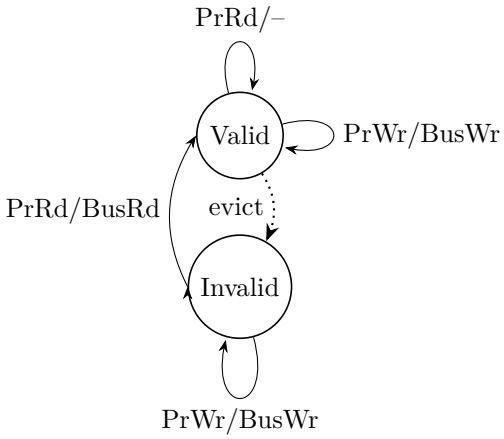
First, what we need to understand is that we are not really talking about the state of the cache, we are talkning about the state of a **cache line**. So let us first about what we need.

We need to know wether or not the piece of data that we have is valid or not. This give us two states: So let us first about what we need.

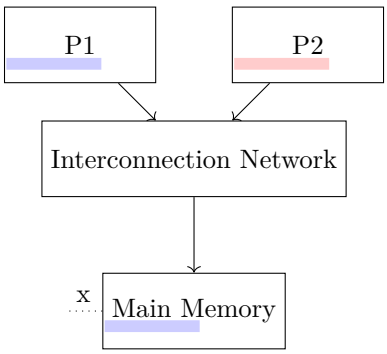
We need to know wether or not the piece of data that we have is valid or not. This give us two states:



But is it a bit strange, how can that be right? We never invalidate anything? We miss eviction here, but when we evict something did means that we have a 'new finite state machine'. Each fsm represent one piece of memory and not the other. So it should be more something like this



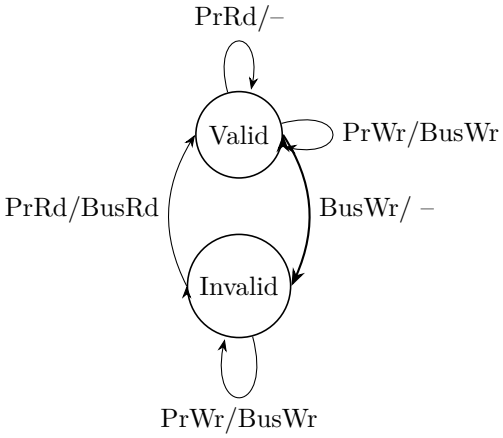
In fact the finite state machine **disappear** when we evict our data. The state that we create here has already been here before, the valid bit in our cache is the state of the fsm. So now the problem at step four look more like this:



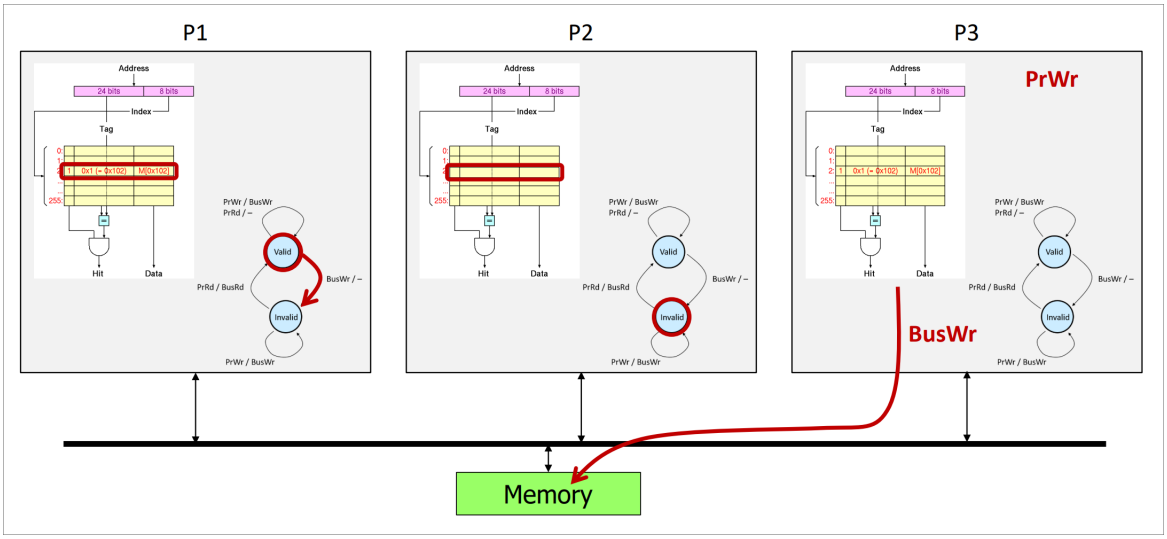
So now what happens is the following:

- Someone is writting to his cache → he then perform a **bus writing**
- So the bus has the information of a writting

So us, as the other processors, we also have access to the bus. Everytime there is a bus write and that it doesn't come from our processor, this means that the value in mean memory has been changed → our value is invalid. So now we have a way of knowing when to change:



Bus this is pretty bad, firstly having one processor being write-through is not very good, this means everytime we are writting to memory, it is a miss. Now imagine this for a multiprocessor. This is even worse than before, maybe even with 2 processors we will have a bottleneck.



Imagine that we have a program with just a for loop `for (int i = 0; i < 10; i++)`. At each `i++` we would need to write to memory even though everybody else doesn't case about our local variables. On the other hand downloading an image, a pdf, etc.. is something important! But as the processor perspective we don't know.

A way of solving this issue is to make a difference for the programmer. We allows the programmer to flush the cache when he needs to transport something to main memory. When he doesn't need to transport, he just doesn't flush → value stays in the cache. But this is not very good, we are adding work to the programmer. Is there a way to solve this issue:

A 3-State Write-Back Invalidation Protocol (MSI)

First what we have done before:

2-State Protocol

- Simple hardware and protocol
- **Bandwidth** (every write goes on bus)

3-State protocol

- **M**odified
  - Only one cache has valid/latest copy
  - Memory is stale, that is the content is not up to date
- **S**hared
  - One or more caches have valid copy
- **I**valid

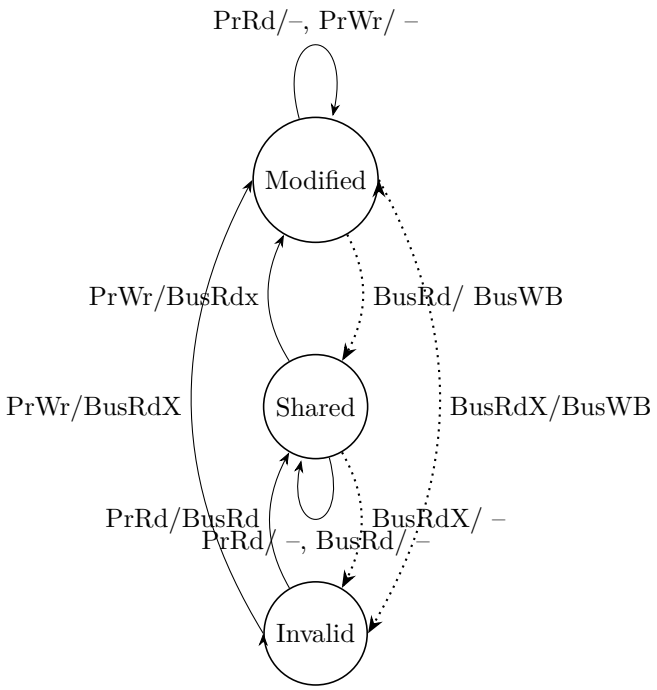
It must invalidate all other copies before entering modified state requires bus transaction (order and invalidate). This requires bus transaction (order and invalidate)
This means that now our finite state machine has three states:

MSI: Processor and Bus Actions

For our processor, we have only two actions: `PrRd` → processor read and `PrWr` → processor write.
On the other hand, for the bus we have:

- `BusRd` → bus read; Read **without intent to modify**, data could come from memory or another cache
- `BusRdX` → bus read exclusive; read **with intent to modify**, must invalidate all other caches copies
- `BusWB` → writeback; cache controller puts contents on bus and memory is updated

Definition 10. cache-to-cache transfer **cache-to-cache transfer** occurs when another cache satisfies **BusRd** or **BusRdX** request with a **BusWB**



Remark 55. An important thing to do is to understand for each arrows, why do we have to do another execution. What when performing a processor write from invalid to modified we have to do a bus read exclusive. Why when we are in invalid and we try to read we also have to read from the bus (this is kind of obvious but you get the intuition for the rest).
For each transition in the MSI protocol, it is important to understand why a bus transaction is needed or not.

- **Processor read in the Invalid state (Invalid → Shared,** `PrRd / BusRd`**)**
  If the cache line is in the Invalid state, the cache does not have the data. Therefore, it must issue a `BusRd` to get the data from memory or from another cache. Since the processor only wants to read the data, the line can enter the Shared state.
- **Processor write in the Invalid state (Invalid → Modified,** `PrWr / BusRdX`**)**
  If the processor wants to write to a cache line that is not present, the cache must first get the data and also make sure no other cache has a copy. A `BusRdX` fetches the data and invalidates all other copies, so the cache can move to the Modified state.
- **Processor write in the Shared state (Shared → Modified,** `PrWr / BusRdX`**)**
  In the Shared state, the same cache line may exist in several caches. Before writing, the cache must ensure it is the only one with a valid copy. Issuing a `BusRdX` invalidates the other copies and allows the transition to the Modified state.
- **Processor read or write in the Modified state (Modified → Modified,** `PrRd / --`**,** `PrWr / --`**)**
  In the Modified state, the cache already has the only valid and up-to-date copy of the data. Therefore, no bus transaction is needed.
- **Bus read while in the Modified state (Modified → Shared,** `BusRd / BusWB`**)**
  If another cache issues a `BusRd`, the modified cache must provide the updated data by writing it back on the bus. After that, the cache no longer has exclusive ownership and moves to the Shared state.
- **Bus read-exclusive while in the Modified state (Modified → Invalid,** `BusRdX / BusWB`**)**
  If another cache requests exclusive access, the modified cache must first write back the updated data and then invalidate its own copy.
- **Bus read-exclusive while in the Shared state (Shared → Invalid,** `BusRdX / --`**)**
  In the Shared state, the data is the same as in memory. When a `BusRdX` is observed, the cache simply invalidates its copy without writing back.

What is the important thing here is that now, we are able to PrWr without having to do any thing else. This is possible because of the dirst/modified state that we added. This way we **know** that we have in our cache is not shared with main memory and therefore might as well do what we want with it (until it is evicted).

Exampe

Now let us take for example the following table of actions:

	CPU Action	P1 state	P2 state	P3 state	Bus Action	Data from
0		- (=|)	- (=|)	- (=|)		
1	P1 reads x					
2	P3 reads x					
3	P3 writes x					
4	P1 reads x					
5	P2 reads x					

Let us do a few cases. For the first one we have a cache miss, this means that we go from invalid → shared. For the other one everybody care about a bus read (in I)

	CPU Action	P1 state	P2 state	P3 state	Bus Action	Data from
0		- (=|)	- (=|)	- (=|)		
1	P1 reads x	S	I	I	BusRd	MEM
2	P3 reads x					
3	P3 writes x					
4	P1 reads x					
5	P2 reads x					

Now p3 reads x, this means that we do exactly the same thing as p1 has done. p1 doesn't care about a bus read has it is alread in S.

	CPU Action	P1 state	P2 state	P3 state	Bus Action	Data from
0		- (=|)	- (=|)	- (=|)		
1	P1 reads x	S	I	I	BusRd	MEM
2	P3 reads x	S	I	S	BusRd	MEM
3	P3 writes x					
4	P1 reads x					
5	P2 reads x					

Now there is something intresting! Now we have a processor write from a shared state $\rightarrow$ we go to the dirty state M and the Bus is doing a BusRdX. As the other state the P1 was in shared but received a BusRdX $\rightarrow$ invalid

	CPU Action	P1 state	P2 state	P3 state	Bus Action	Data from
0		- (=|)	- (=|)	- (=|)		
1	P1 reads x	S	I	I	BusRd	MEM
2	P3 reads x	S	I	S	BusRd	MEM
3	P3 writes x	I	I	M	BusRdX	MEM
4	P1 reads x					
5	P2 reads x					

For the fourth step, P1 now reads the new value which makes up to date again. And now the P3 is now also up to date with everybody $\rightarrow$ shared:

	CPU Action	P1 state	P2 state	P3 state	Bus Action	Data from
0		- (=|)	- (=|)	- (=|)		
1	P1 reads x	S	I	I	BusRd	MEM
2	P3 reads x	S	I	S	BusRd	MEM
3	P3 writes x	I	I	M	BusRdX	MEM
4	P1 reads x	S	I	S	BusRd	P3
5	P2 reads x					

For the last steps, this is just a classical reads which make P2 goes into the shared state:

	CPU Action	P1 state	P2 state	P3 state	Bus Action	Data from
0		- (=|)	- (=|)	- (=|)		
1	P1 reads x	S	I	I	BusRd	MEM
2	P3 reads x	S	I	S	BusRd	MEM
3	P3 writes x	I	I	M	BusRdX	MEM
4	P1 reads x	S	I	S	BusRd	P3
5	P2 reads x	S	S	S	BusRd	MEM

this is pretty nice, but this is not perfect. Because of that, there many other similar protocols that exists. For instance: **4-State (MESI) Invalidation Protocol**

Often called the **Illinois** protocol

- **M**odified (dirty)
- **E**xclusive (unshared clean = only copy, not dirty)
- **S**hared
- **I**nvalid

This requires **shared** signal to detect if other caches have a copy of block. Also cache flush for cache-to-cache transfers

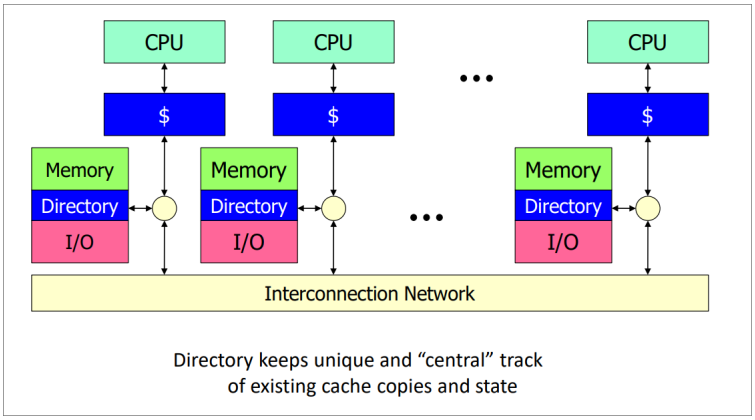
- Only one can do it though

There is other protocol with more states, different transactions etc... There is a lots of research on how to minimize the coherence traffic (=better detect when it is not necessary)

An important problem is scalability beyond a few processors/cores → **Directory-Based Cache coherence**

Directory-Based Cache Coherence

Imagine a system like this:



Imagine now that there is a slow interconnection netwrok.

Directory-based cache coherence is a technique used to make cache coherence scalable when the number of processors increases.

Instead of broadcasting coherence requests to all caches (as in snooping-based protocols), the system maintains a **directory** that keeps track of which caches have a copy of each memory block.

For each memory block, the directory stores:

- The state of the block (e.g., shared, modified)
- The list of caches that currently hold a copy

When a processor wants to read or write a block, it sends a request to the directory. The directory then:

- Sends the data if no conflict exists
- Or sends invalidation or update messages only to the caches that actually have a copy

This avoids broadcast traffic on the interconnection network and greatly reduces coherence traffic, making the protocol scalable to many processors, especially when the network is slow.

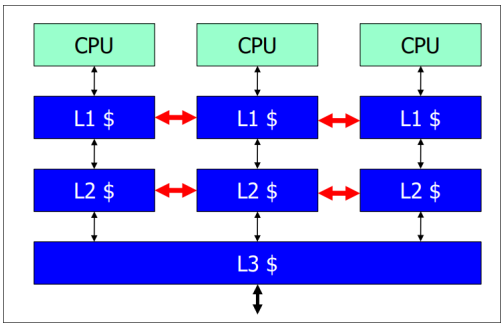
Directory-based coherence is therefore preferred in large multicore and multiprocessor systems.

Snoopy vs. Directory-Based

- **snoopy protocols** are distributed coherence protocols at the cache level
  - Scalability issues
  - Unnecessary coherence traffic
  - Fast
- **Directory-based protocols** are centralized protocols at the memory level
  - Scalable
  - Coherence traffic only as actually needed
  - Latency isues, due to centralization
  - Granularity issues, linked to latency issues

Multilevel Caches

The other problem is that we don't only have caches, we have **multiple level of caches**.
What happens if instead of CPU → cache → Mem, we have CPU → L1 cache → level 2 cache → level 3 cache → Mem?
What we would need to do is to make each cache level speaks to themselve, we could replicate snooping at all levels.



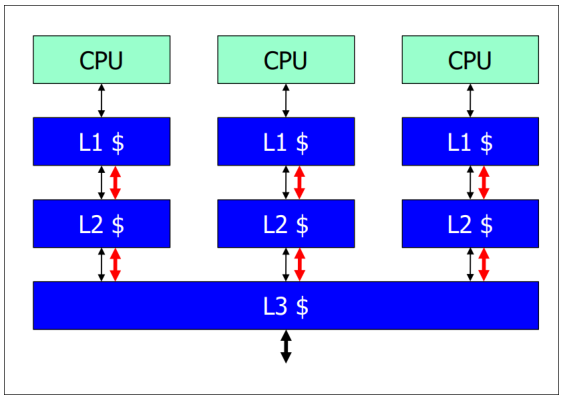
Inclusion Property between Caches at levels n-1 and n

1. **Same content**: Content of L(n-1) cache is always a subset of the corresponding L(n)
   A bus transaction an L2 cache is also always relevent for the L3 cache, hence a **snoop at L3 is sufficient**
2. **Same state**: If a cache line is marked as owned/modified in L(n-1) cache, then it should also be so marked in the corresponding L(n) cache
   If a bus transaction requests a cache line in owned/modified state in L3, the **L3 cache can determine this on its own without checking L2**

So there is no need of snooping any more?

Is Inclusion Naturally Maintained?

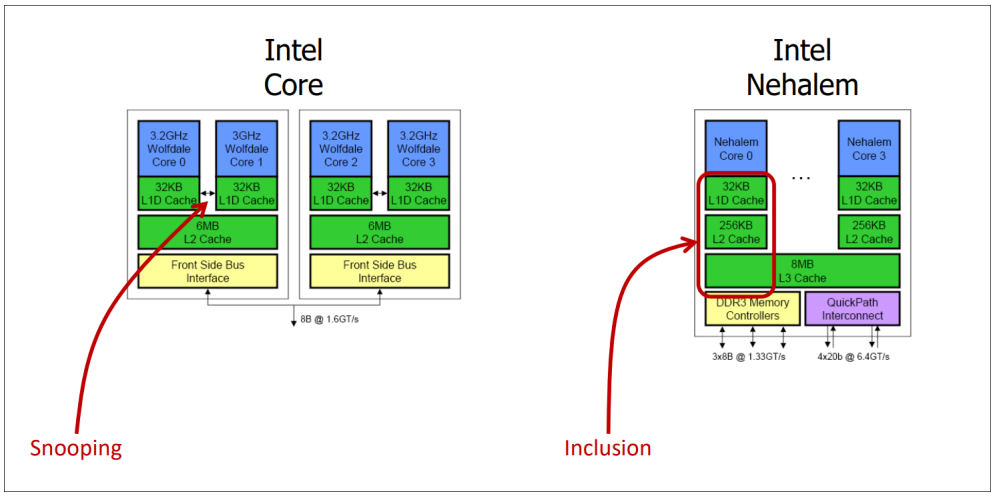
In some cases yes: In a Miss, it is, as L1 misses go to L2 and eventually the data will be in both. However, in the general case **no**: for instance, L2 will eventually decide to evict a given line and that may be also in L1 → if nothing special is done, **inclusion will be violated**
What we need is to **propagate invalidations** and **evictions** up the hierarchy to keep, for instance, L1 informed of what happens in L2.



Remark 56. Which one is easier between snooping and maintaining inclusion is not trivially determined and well beyond this course...

Intel core and Intel Nehalem

So if we take a look of what happen on commercial processor, it depends:



5.2 Examples of Cache Coherence

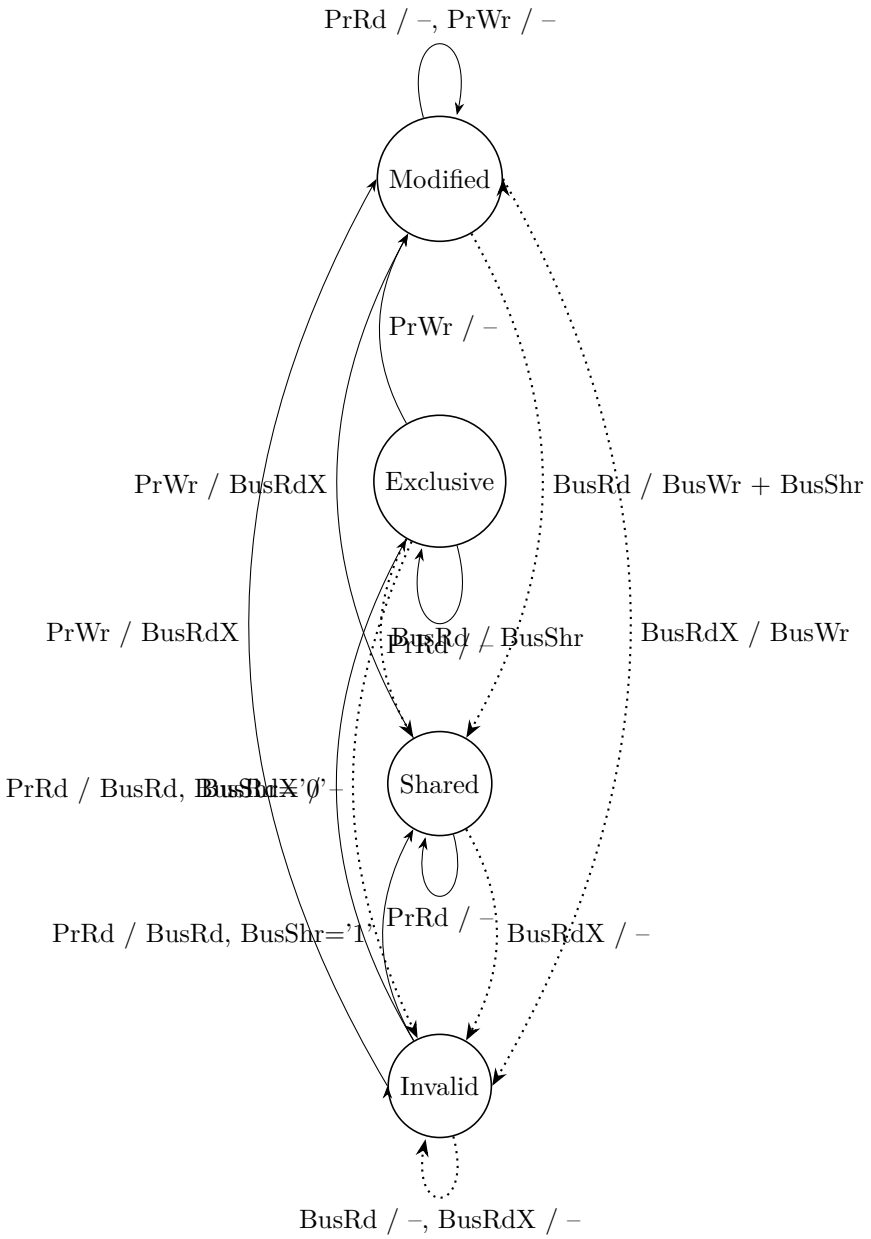
So now we will try to simulate how what we have seen in the previous section is working concretely. For this we will use the **MESI Protocol** **MESI Protocol**

Consider a **system of four processors** P0 to P3, each having its own cache C0 to C3 respectively. The **MESI protocol** is used to maintain coherence among the different caches.

With this protocol, a cache line can be in one of four states:

- Modified
- Exclusive
- Shared
- Invalid

Let us take a look at the finite state machine of the following protocol (the dotted lines are the snooping lines)



The difference is now that we can be exclusive dirty (the modified state) **or** exclusive clean. So intuitively we have:

- `Invalid` → "I do not have a valid copy of X"
  - The cache line is empty or outdated
  - Any read/write needs to fetch data from elsewhere
- `Shared` → "I have a clean copy of X, and **other caches may also have it**"
  - Same value exists in memory
  - Multiple caches may hold X
  - **Read-only** state
  - Writing is **not allowed** without invalidating others
- `Exclusive` → "I have the **only** copy of X, and it matches memory"

  - No other cache has X
  - Memory is up to date
  - Can silently upgrade to Modified on write
- Modified → "I have the only copy of X, and memory is outdated"
  - This cache has changed the value
  - Memory must be updated before other can read
  - **Dirty** data

Without the **Exclusive** state a read would always go:

$$\text{Invalid} \rightarrow \text{Shared} \rightarrow \text{Modified}$$

The MESI optimizes it to:

$$\text{Invalid} \rightarrow \text{Exclusive} \rightarrow \text{Modified}$$

MESI Protocol Shared and Exclusive States

In the MESI protocol, the following needs to be considered when a processor reads a data item that was previously not in tis own cache:

- If the data is already stored in another processor cache, then the status of the cache line holding this data is set to the **Shared** state
- If the data is not already stored in any other cache, then the status of the cache line which will hold this data is set to the **Exclusive** state.

MESI Protocol BusRd and BusShr

To know if the data is already in another cache or not, when one processor reads a data item, with a BusRd transaction, the other caches must indicate if they hold a copy of the data item by setting a special bus signal called BusShr to '1'
If the signal is '0', we know that the particular data item is not held in any other cache

Part I

- In the following diagrams, you are given the current state (labeled as **Old**) of a specific cache line in the four caches C0 to C3
- For each cache, the address of the data saved in that cache line, as well as its state, is specified
- When one of the processors (P0 to P3) executes an instruction that requires a memory access, then a state change is triggered in one or more of the caches (C0 to C3)
- This change is shown, for one cache, under the label **New**

We are now required to fill the rest of the template as follows:

- Specify the new state/address of the remaining caches
- Specify the memory acess that caused these changes; your answer should be in the form

  P<n>:Load/Store<address>

  Where P<n> is the processor that executed the memory operation, Load/Store is the type of memory access and <address> is the address of the data to be loaded/stored

Note that there might be up to 3 possible answers for each case and that **you are required to list all possible answers** in the provided space

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	I	0x1000	S	0x1000	S	0x2000	S	0x1000	
New	S	0x1000							
	S	0x1000							
	S	0x1000							

The only way here to go from Invalid to shared for the C0 cache is by doing a processor read → P0:Load 0x100. This way all the other cache go from shared to shared still.

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	I	0x1000	S	0x1000	S	0x2000	S	0x1000	
New	S	0x1000	S	0x1000	S	0x2000	s	0x1000	
	S	0x1000							
	S	0x1000							

So now here is a couple of examples:

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	E	0x1000	I	0x1000	S	0x2000	I	0x1000	
New	S	0x1000							
	S	0x1000							
	S	0x1000							

So here we go from exclusive to shared. The only way of doing that is a BusRd, now we go in our finite state machine diagram and we look for all the possible way of lauching a BusRd. For us, we will chosse P1:Load 0x1000 This way we have, the only way for the C0 to go to the shared state, is to have another processor launch a BusRd, The way of doing that is for instance having the P1 to read the value at 0x1000:

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	E	0x1000	I	0x1000	S	0x2000	I	0x1000	
New	S	0x1000	S	0x1000	S	0x2000	I	–	P1:Load 0x1000
	S	0x1000							
	S	0x1000							

As we can go the same thing from the third processor (both are totaly the same). And as everyone will be in the shared state, someone else would also need to read from memory P2.
As the C3 processor as the same state as the C1 processor, we also have the `P3:Load 0x1000`.
If we take a look at the C2 cache it is shared **but for the address 0x2000**. this totally equivalent as being invalid for 0x1000 right? So might as well do the same thing as the other:

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	E	0x1000	I	0x1000	S	0x2000	I	0x1000	
New	S	0x1000	S	0x1000	S	0x2000	I	–	P1: Load 0x1000
	S	0x1000	I	0x1000	S	0x1000	I	0x1000	P2: Load 0x1000
	S	0x1000	I	0x1000	S	0x2000	S	0x1000	P3: Load 0x1000

Part I Case 2

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	I	0x1400	I	0x1200	I	0x1200	M	0x1200	
New							I	0x1200	
							I	0x1200	
	S	0x1000					I	0x1200	

So for this one, we need to go form the Modified state to the invalid state. To do so, we need a BusRdX signal from another processor. To do so, we can do a processor write from an invalid state.

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	I	0x1400	I	0x1200	I	0x1200	M	0x1200	
New	M	0x1200	I	0x1200	I	0x1200	I	0x1200	P0: store 0x1200
	I	0x1400	M	0x1200	I	0x1200	I	0x1200	P1: Store 0x1200
	I	0x1400	I	0x1200	M	0x1200	I	0x1200	P2: Store 0x1200

Case 3

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	M	0x1300	I	0x1200	I	0x1100	E	0x1100	
New 0x110									M
							M	0x1100	
							M	0x1100	

So the only way to go from the Exclusive clean state to the dirty state is to write something to our value, making it dirty → PrWr.

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	M	0x1300	I	0x1200	I	0x1100	E	0x1100	
New	M	1300	I	0x1200	I	0x1100	M	0x1100	P3: Store 0x1100
							M	0x1100	
							M	0x1100	

Case 4

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	I	0x1200	M	0x1200	S	0x1000	S	0x1000	
New					M	0x1000			
					M	0x1000			
					M	0x1000			

As before because we are going from a clean state to a dirty state → we need to write something. The difference here is how would that impact other cache. This time we are putting a BusRdX on the bus which means, if we have it, you may care to know that it is tail. The 0x1200 address doesn't care so they don't change. As for the C3 cache we do care here, our value has been changed now and what we have in our cache is completely useless now... → we go to invalid state:

	Cache								P<n>: L/S <addr>
	Cache C0		Cache C1		Cache C2		Cache C3		
	State	Address	State	Address	State	Address	State	Address	
Old	I	0x1200	M	0x1200	M	0x1000	S	0x1000	
New	I	—-	M	0x1200	M	0x1000	I	—-	P2: Store 1000
					M	0x1000			
					M	0x1000			

Donc il y a en a jusqu'a 11, je vais pas tous les copier ici parce que voila... mais ducoup dans le pdf *5.b Examples of Cache Coherence.*

5.3 Memory Consistency

Before diving into consistency, what is the difference between coherence and consistency?

Coherence

- **What values** should be returned by a set of reads and writes to a **single address**
- An issue of data integrity; if violated, I get wrong data

Consistency

- **When** written values will be returned by reads to **multiple addresses**

- Not an issue of data integrity: data are correct from the perspective of a single processors, but observations from different processors may show different orderings and **contradict the expecation of the programmer**

So if for instance we take two prgrams Where each one runs on a different thread:

Processor or Thread #1

```
A = 0;
...
A = 1;

if (B == 0) ...
```

Processor or Thread #2

```
B = 0;
...
B = 1;

if (A == 0) ...
```

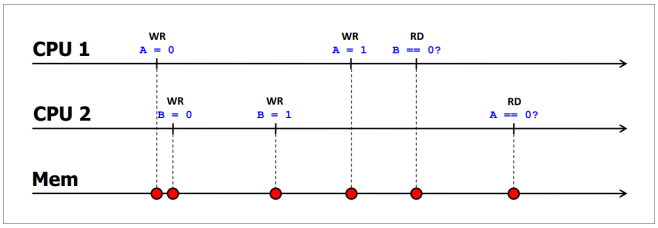
So can both tests be true? Not really right? If we were able to run those two program in parallel this would even be **wrong**.

Note that this is not a problem of accesses to a single location (coherence) but of the interaction of **several accesses to more than one location**.

It is also the program of **when** a new value must be visible.

Ideally: Strict Consistency

What we would want is a perfect world would be that everything happens in memory **exaclty in the order it has been issued**.



However in any distributed system is (virtually) **impossible to obtain a gloval time**, hence let's forget about it...

The minimum that we need is to have the same clock for all the CPU. And if we take what we have seen in cs-173, this is a hard thing to do to: *conduct* the clock to everyone. But this is not even the case for most of distributed system, most of them doesn't even have the same clock.

Remark 57. Even on this day our laptop core doesn't have the same clock

Because we cannot have a perfect notion of time for everyone, let us relax:

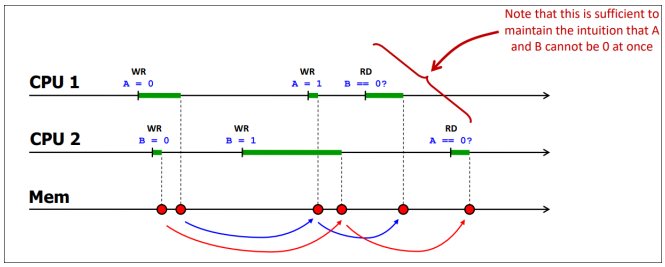
Definition 11. Sequential Consistency The result of any execution is as if:

the operations of each individual processor were executed the order specified by its program, and

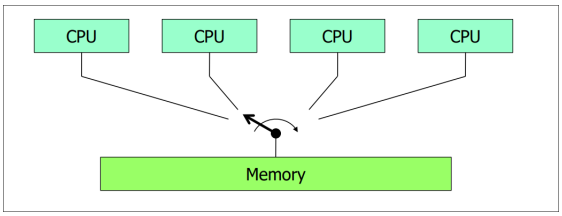
- the operations of the different processors were arbitrarily interleaved

More Practical: Sequence Consistency

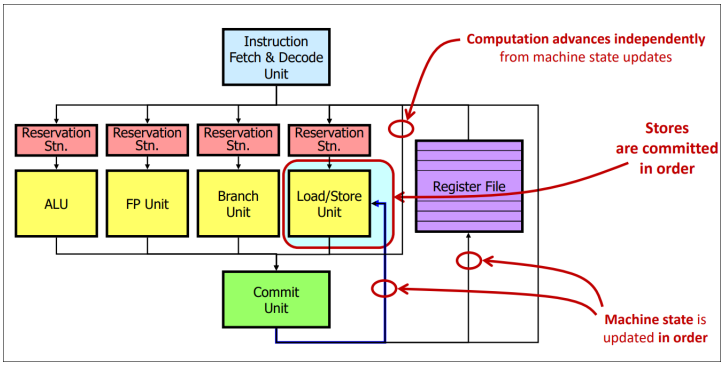
The idea is to let some time after the write/read for the other processors/memory to *process* the new value



A more formal way of seeing it is:



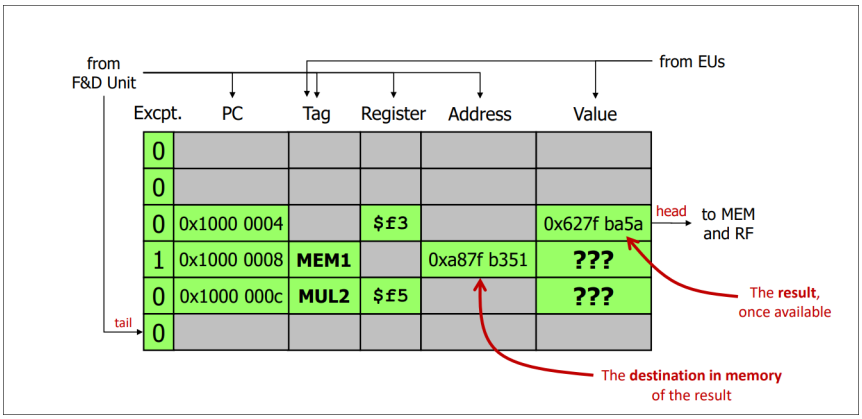
Every CPU have access to memory when they are connected with a switch, they do what there **load and store** in program order, everything is atomic (no other memory operation is started while the previous has not completed)
After every memory operation, the **switch is randomly changed** ***load and store*** *in program order*???? But remember this:



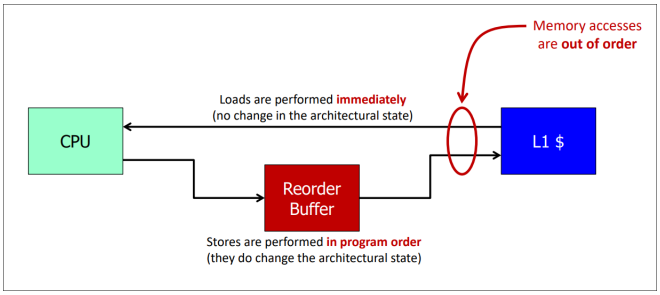
Now we have three possibilities (this is what we have done in the previous sections):

1. We keep all the memory accesses by whatever we know until now
2. We keep all our addresses or all our accesses in order
3. We keep someting in order

The correct answer is the last one, keeping some of the thing in order. We had to do so to solve the memory dependency issue. First we needed in the first place to reorder instruction at writeback:



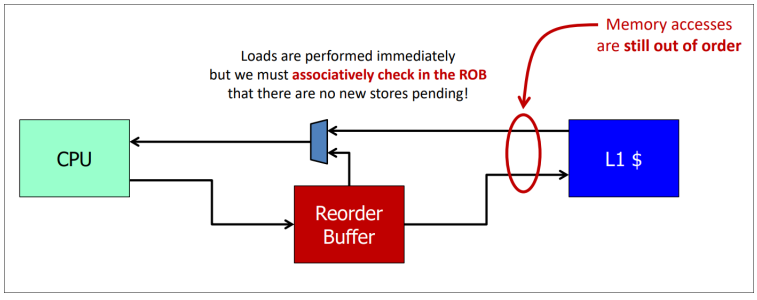
Exception was the reason why we needed to do so. In fact our memory path looks more like this:



But there is a **mistake**, we have badly violated consistency even on a uniprocessor: the relative order of read and writes is now possibly wrong. If we write at `0x1000`, we read at `0x1000` then we have an issue, we have read the previous value of `0x1000` which is now deprecated.

Correct Actual Memory Path

What we need to check if while we are waiting to write our value, it is in the Reorder Buffer.



Now we do honour RAW dependencies and **uniprocessor consistency** is correclty implemented
Still, other processors do not have such a bypass...
Uniprocessor memory accesses our of order → no sequential consistency

Dependences through Memory

The way we detect and resolve dependences through memory (a store at some address and a subsequent load from the same address) is the same as for registers.
For every load, check the ROB

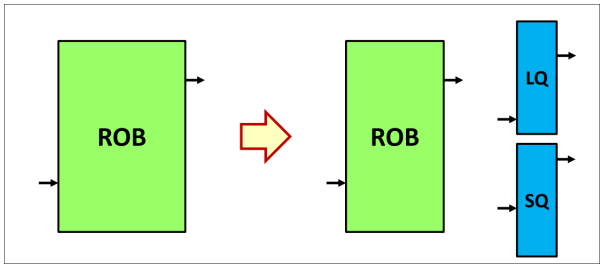
1. If there is **no store to the same address** in the ROB, get the value from memory (i.e., from the cache)
2. If there is a **store to the same address** in the ROB, either get the value (if ready) or the tag

but there is an additional situation now

3. If there is a store to an **unknown address** in the ROB or if the address of the load is unknown, **wait**

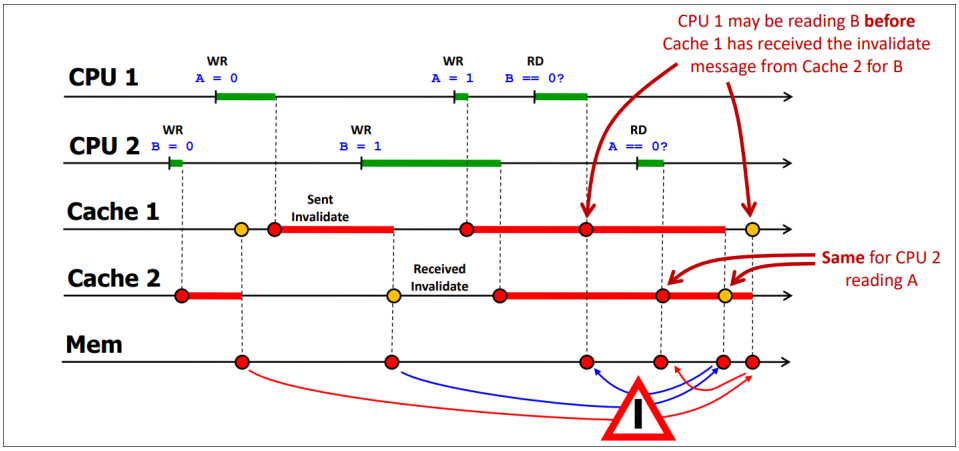
Load-Store Qeues

In practice, the **memory part of the ROB** is implemented separately and is called a **Load-Store Qeue** (in turn, usually implemented as a Load and Store queues)



More Challenges to Sequential Consistency

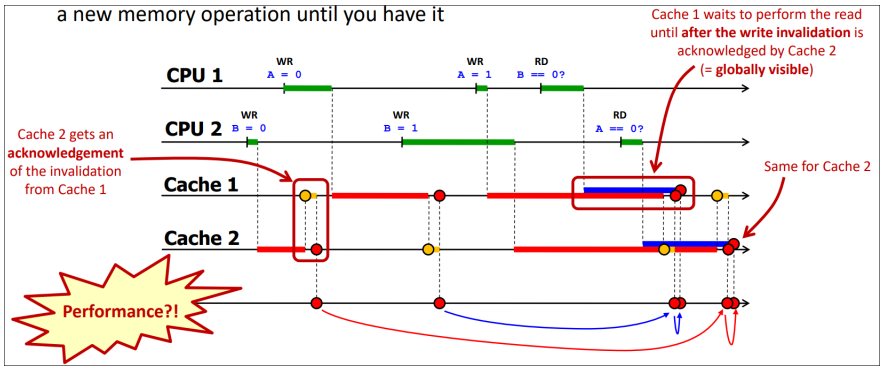
Consider a normal system with caches and using a simple invalidate snooping protocol.



The issue that we have is **not coherence** at some time we **will have the correct** value in both caches. The problem we have is only the fact that our instruction are overlapping each other. The problem is a **when**.

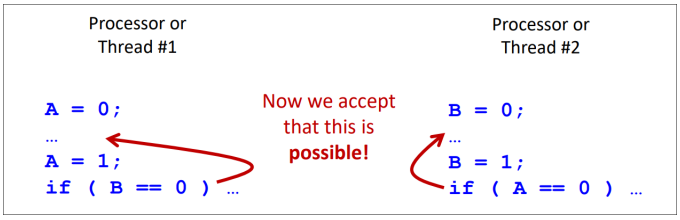
how to get Sequential Consistency

A naive way for us to solve this is **waiting**: wait for the acknowledgement of the invalidation and do not start a new memory operation until you have it



But this is bad, maybe there is a better way of doing it

Relaxing Write → Read order: Proeccors Consistency Model



We **now admit that both tests can be true at once** (Read advances over independent Write)
Still, we **enforce write order so that the values of A and B will eventually be both 1** (in some unspecified future)
IA-32 and some other processor (IBM 370) implement this model

Many Relaxed Consistency Models

The goal it to choose consistency models which are **efficiently implementable byt do no 'surprise too much' programmer**
Different combination on **what can be disordered** (W → R, W → W, R → W, R → R, ...) and other details

- Wisconsin/Stanford processor consistency
- IBM 370
- Intel IA-32
- Sun Total Store Order
- USC/Rice weak ordering
- Stanford release consistency
- DEC Alpha
- IBM PowerPC
- Sun's Relaxed Memory Order
- ...

Only system programmers (OS, libraries, middleware) typically see these details and act on them to implement higher level functions, uniforms across all or most systems

Relax Everything: Release Consistency Model

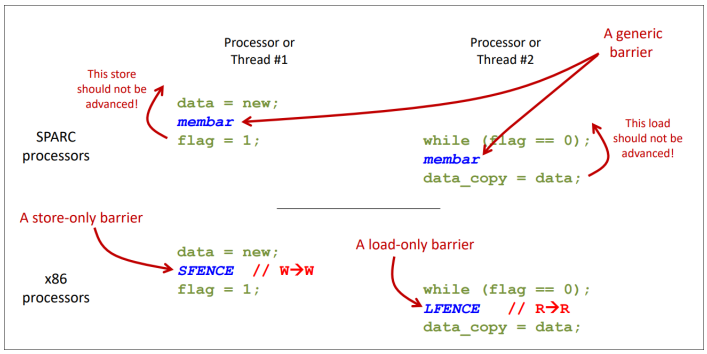
Still **honour every dependence locally** in a processor, but **otherwise co,pletely disregard ordering** accross normal loads and stores
Introduce special synchronization operations that have strict ordering

- Typically some instruction are used to **acquire access** (S_A) to a shared variable and enforce the orderings $S_A \to$ W and $S_A \to$ R while other instructions are used to **realease access** (S_R) to a shared variable and enforcethe orderings W $\to S_R$ and R $\to S_R$
- Another approach is to have memory **barriers** or **fences** (**S**) that act like $S_A + S_R$ and enforce all orderings W $\to$ S, R $\to$ S, S $\to$ W, and S $\to$ R (i.e., the execution of a memory barrier waits for all pending loads and stores to **complete and be globally visible**, and does not let any successive load or store start).

Put the **burden on the programmer/compiler** and be as aggressive as you can in the hardware

Memory Barriers/ Fences

The `membar` guarantee that everything that is before has gone everywhere by the time it stops executing.



Remark 58. Remember to take a long look at this to make sure I understand

Atomic Instructions

Now if we take a look at the other way of solving this issue:
Atomic instructions is a combination of load and store **without interference from others**. This is a typical way to implement **acquire access**

- **Test-and-set**: interchanges a fixed value for a value in memory
- **Atomic exchange** or **swap**: interchanges a value in a register for a value in memory
- **Copare-and-swap**: compare a register value to a value in memory addressed by another register, and if they are equal, then swap a thirf register value with the one in memory.

The **compare and swap** is **good** because it writes only if the comparison is succesful. It is **bad** because it needs **three source** registers

Consistency is Hard

Memory **Consistency is hard**, there is subtle interaction between hardware optimizations (e.g., sotre buffers, reordering) and memory models make **reasoning about correctness challenging**
The code is **sublty processor dependent**

- Programs can **behave differently** based on the processor's memory consistency model (e.g., x86 vs ARM), requiring careful design portability.

Simplified for software programmers: to shield developers, **consistency mechanism are encapsulated** in:

- system libraries (e.g., synchronization primitives, atomics)
- APIs (e.g., C++ `std::atomic`,pthreads, java volatile)

These APIs are simple, intuitive, and **uniform across platforms** while hiding processor-specific details

5.3.1 Summary Multiprocessors

- Multiprocessors have come to the consumer market and are here to stay
- Peculiar multiprocessors (e.g., heterogeneous) have been for many years in high-end embedded systems
- They can usually take advantage of most of the progress in uniprocessor design and performance optimization
- Yet, they involve major challenges when it comes to preserve the multithreaded performance of uniprocessors (interconnection, coherence, consistency, etc.)
- Scalability is one of the greatest architectural issues of the future

Chapter 6

Hardware Secrurity

6.0.1 Why Hardware Security

Software complexity

- OSes and hypervisors are too complex to be trusted to be bug free
- Who can trus OSes and hypervisors?! **Secure processor architextures**

Microarchitectural side-channel attacks

Sharing with other users gives them the ability to discover our secrets

- Shared caches, shared processors (branch predictors, pipelines, etc.) ← cs-200

Physical monitoring attacks and physical side-channell attacks

Users cannot physically protect their computing Hardware

- Hardware is often in the cloud
- Hardware is embedded and remote (Internet of Things, IoT)

Outline of This Lecture

- Basic Definitions
- Attacks on Memory to Compromise Integrity (Rowhammer)
- Covert Channels and Side-Channel Attacks
- Attacks on Timing to Break Isolation and Confidentiality (Timing Side-Channel Attack)
- Attacks on Memory to Break Isolation and Confidentiality (Cache Side-Channel Attacks)
- Combined Attacks to Break Isolation and Confidentiality (Meltdown)
- Combined Attacks to Break Isolation and Confidentiality (Spectre)

Basic Definitions

Specification of the threats that a system is protexted against:

- **Trusted Computing Base**: what is the set of trusted hardware and software components
- **Security properties**: what the trusted computin base is supposed to guarantee
- **Attacker assumptions**: what a potential attacker is assumed capable of
- **Potential vulnerabilities**: what an attacker might be able to gain

Classic Security Properties

- **Confidentiality** → prevent the disclosuer of secret information
- **Integrity** → prevent the modification of protected information
- **Availability** → guarantee the availability of services and systems

We will also speak of **isolation**, that is the possibility to prevent any interaction between users and processes, often used to guarantee confidentiality and integrity

Attacks on Memory to Compromise Integrity

y **Dynamix Random-Access Memory**

- DRAMs are the densest (and thus cheapest) form of random-access semiconductor memory
- DRAMs store **information as charge in small capacitors** part of the memory cell
- Firat patented in 1968 by Robert Dennard, scaled amazingly over decades and was somehow an important ingredient of the progress of computing systems
- Charge **leaks off** THE capacitors due to parasitic resistances → every DRAM cell needs a **periodic refresh** (e.g., every 60ms) lest it forgets information.

Apparently only a Reliabilty Issue

- To increase density (i.e., reduce cost) memory cells have becom incredibly small (→ **small storage capacitance, smaller noise margine**) and word line go extremely clode to each other (→ **larger crosstalk capacitive coupling**)
- Frequent **activation of word lines** neighbouring particular cells between refreshes may **flip the cell states** due to various forms of capacitive coupling
- **Disturbance errors** have been known design issue of DRAMs sinve ever, but failure in commercial DD3 chips was demonstrated in 2014

Rowhammer

For instance let us took a remarkably simple code

```
codela:
        mov (x), %eax // read from address x
        mov (y), %eax // read from address y
        clflush (x)
        clflush (y)
        mfence
        jump codela
```

- `mov` instructions activate neighbouring rows

- `clflush` unprivileged x86 instructions flush the cache from the value of X and Y (so that future accesses are misses) and `mfence` roughly waits for the flush
- Repeat as quickly as possible

An opportunity for Attacks

Rowhammer effectively violates memory protection ("if I can read, I can also write") which is a key ingredient to privilege separation across processes.
By accessing locations in neighbouring rows one could gain unrestricted memory access and privilege escalation

- Allocate large chunks of memory, try many addresses, learn weak cells
- Release memory to the OS
- Repeatedly map a file with RW permissions to fill memory with page table entries (PTEs)
- Use Rowhammer to flip (semirandomly) a bit in one of these PTEs; it will now point to the wrong physical page
- Chances are that this physical page contains PTEs too, so now accessing that particular mapping of the file (RW) actually modifies the PTEs, not the file
- Attacker can arbitrarily change PTEs and memory protection is gone

Not that simple in practive, tons of difficulties, but people managed to make it work!

An aside on DRAMs: Data remanence

- A completely different problem with storing data on capacitors: cells may leak information quickly in the worst case but very many do not leak much in typical conditions
- Lowering significantly the device temperature (e.g., use spray refrigerants) makes most cells retain charge for long time (seconds to minutes)
- **Coldboot** attacks:
  - Cool a working DRAM device
  - switch off
  - Move the device to another computer or reboot a malicous OS
  - Read content (password, secret keys)

6.0.2 Covert Channels and Side-Channel Attacks

Covert Channels

- "A covert channel is an **intentional communication** between a sender and a receiver **via medium not designed to be a communication channel**" (Szefet, 2019)
- If we isolate a critical process inside a virtual machine, a covert channel may allow a rogue programme inside of the isolated process (a **Trojan horse**) to leak a secret to some malicious receiver without anyone to notice (no conventional communication channel visible)

Covert Channels vs Side Channels

- **Covert channels**:
  - Intentional communication between a sender and a receiver
  - Both parties collaborate to leak information

- **Side channels**:
  - The victim is unaware of the information leakage
  - The attacker passively observes side effects

Types of Covert and Side Channels

- **Microarchitectural channels**
  - Based on microarchitectural state that is not architecturally visible
  - Architectural state is defined and protected (except for bugs)
  - Exploit shared hardware resources
  - Examples:
    - caches
    - branch predictors
    - pipelines
- **Physical channels**
  - Based on the physical nature of computing systems
  - Examples:
    - power consumption
    - electromagnetic emissions
    - temperature

Timing Side-Channel Attacks latex Copy code

6.0.3 Attacks on Timing to Break Isolation and Confidentiality

Timing Side-Channel Attacks

- Execution time may reveal information about secret data
- Time variability may depend on:
  - input data values
  - execution paths
  - microarchitectural effects (caches, virtual memory, instruction scheduling)

Example: String Comparison

A naïve implementation returns as soon as a difference is detected, revealing the position of the first incorrect character.
A **constant-time** implementation always processes the entire input before returning.

Blinding through Constant Time

- Writing constant-time code is difficult
- Compiler optimizations may reintroduce timing variability
- Even branch-free code may leak information due to:
  - caches
  - virtual memory
  - instruction latency
- Many of the attacks discussed later are ultimately timing attacks

6.0.4 Attacks on Memory to Break Isolation and Confidentiality

Cache Side-Channel Attacks

- One of the oldest and most powerful microarchitectural side channels
- Known since the early 1990s, fully demonstrated in 2005
- Cache state is shared but not architecturally visible
- Attackers distinguish cache hits and misses via high-resolution timing

General Principle

- Victim memory accesses depend on secret data
- These accesses modify the cache state
- The attacker infers secrets by observing cache behavior

Prime + Probe

- **Prime**: the attacker fills cache sets with its own data
- The victim executes and evicts some cache lines
- **Probe**: the attacker reloads its data and measures access time
- Slower accesses indicate victim activity in the corresponding cache set

Advantages of Prime + Probe

- The attacker measures only its own code
- Better control over noise
- Works without shared memory

6.0.5 Combined Attacks to Break Isolation and Confidentiality (Meltdown)

Meltdown

- Catastrophic attack allowing reads of protected kernel memory
- Caused by microarchitectural implementation choices
- Exploits a race between memory access and permission checks
- Some architectures are accidentally resistant

Attack Principle

- The attacker performs a forbidden memory access
- The value is transiently loaded during speculative execution
- The value is used to perform a legal memory access
- This access leaves observable cache traces
- A cache side-channel attack reveals the secret

Affected Processors

- Intel x86: most processors since 1995
- AMD x86: not affected
- ARM, POWER: some implementations affected

6.0.6 Combined Attacks to Break Isolation and Confidentiality (Spectre)

Spectre

- Exploits speculative execution and branch prediction
- Branch predictors are not fully isolated across contexts
- Speculation does not affect architectural state
- Microarchitectural side effects leak information

Attack Principle

- The attacker trains the branch predictor
- A misprediction forces speculative execution of unsafe code
- Speculative memory accesses depend on secret data
- Cache side effects remain after speculation is squashed
- The attacker recovers the secret via a cache attack