Computer Architecture
Prof. Paolo Ienne

Arthur Herbette

October 2025

Contents

Chapter 1

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.

1.0.1 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

Chapter 2

Processors and instruction set architecture

2.1 Instruction set architecture

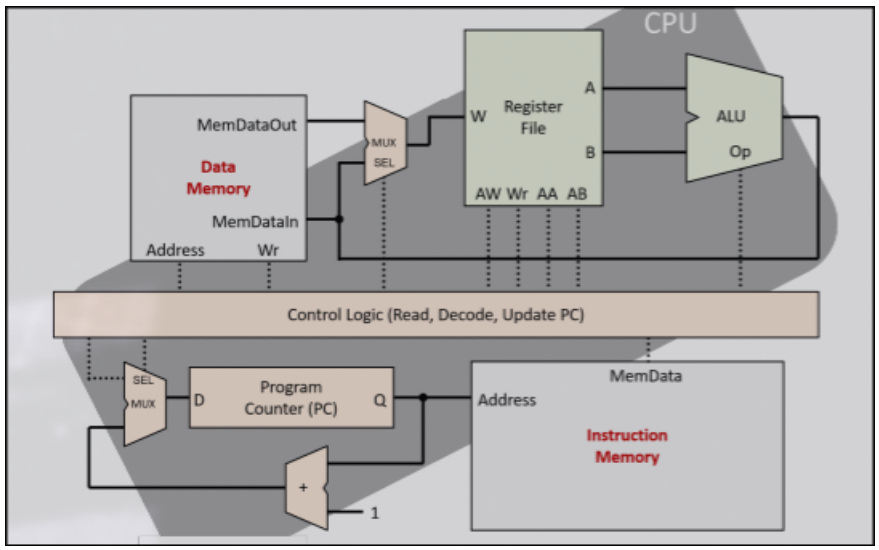
The goal for the beggining of this course is to go from "high-level" perspective to the bottom of the iceberg. First let us look at a piece of code (`c`):

```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);
```

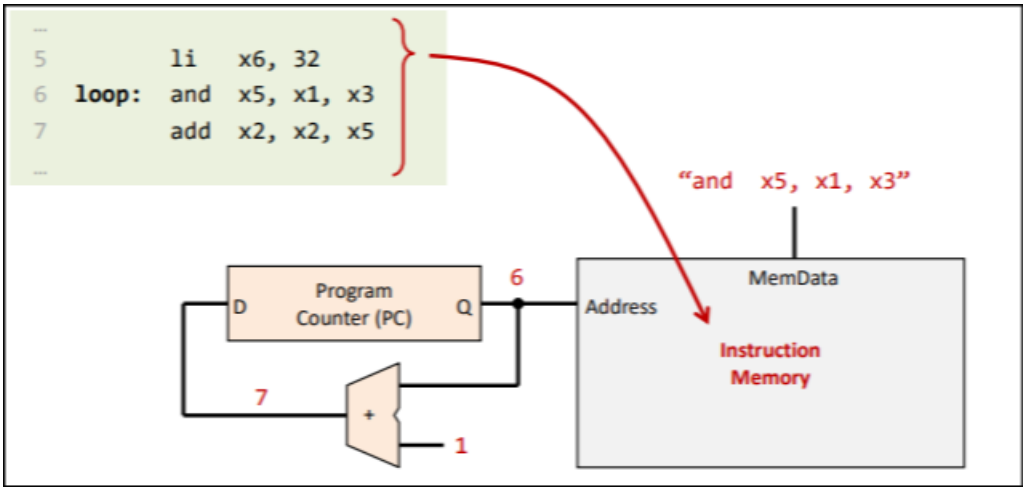
Here we can see that we have variable with expressive names (that we can choose). each variable has a type, the computation we are doing `result + temp` looks like a mathematic formula, the control flow we are using is very intuitive.
In the case of those high-level language, we have an "unlimited" number 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
```

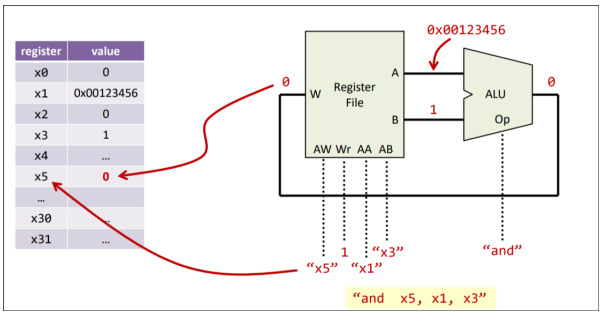
As we can see we have a much more rigid format: we really have a sequence of numbered instructions that is executed line by line. For each instructions, we have an *opcode* that defines the effect of the instruction. Each *variable* has a fixed name and we only have one form of control flow. The question 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

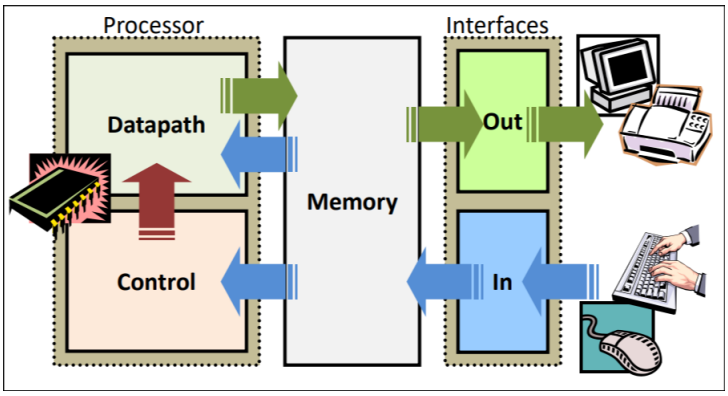


After that the instruction has been fetch, it is processed in the Control logic and then read/write etc... into the register file, and give the information (the opcode) to the ALU for it to know which operation to perform.

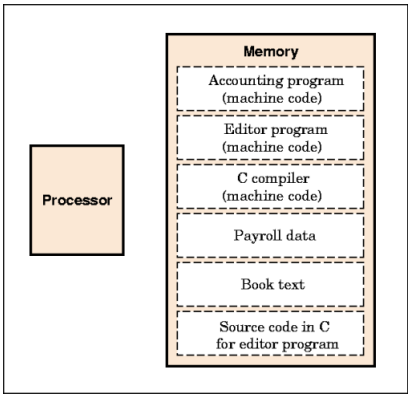


The five classic components of a computer

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).



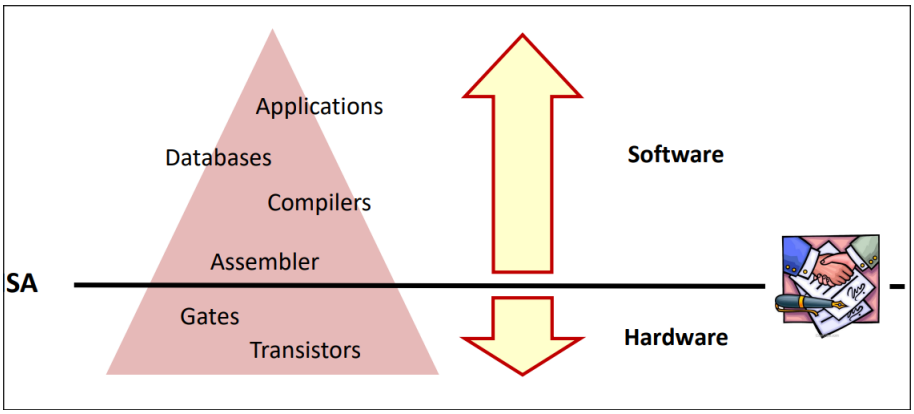
Now a good question to have is: how to decode and encode those instructions and in the mean time, also what makes a good encoding?
A good encoding would be one that allows us two minimize the ressource in hardware and also is

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.

2.2 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
...
```

2.2.1 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.

2.2.2 The stack

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

Définition 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	—

2.3 Memory and Addressing Modes

2.3.1 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:

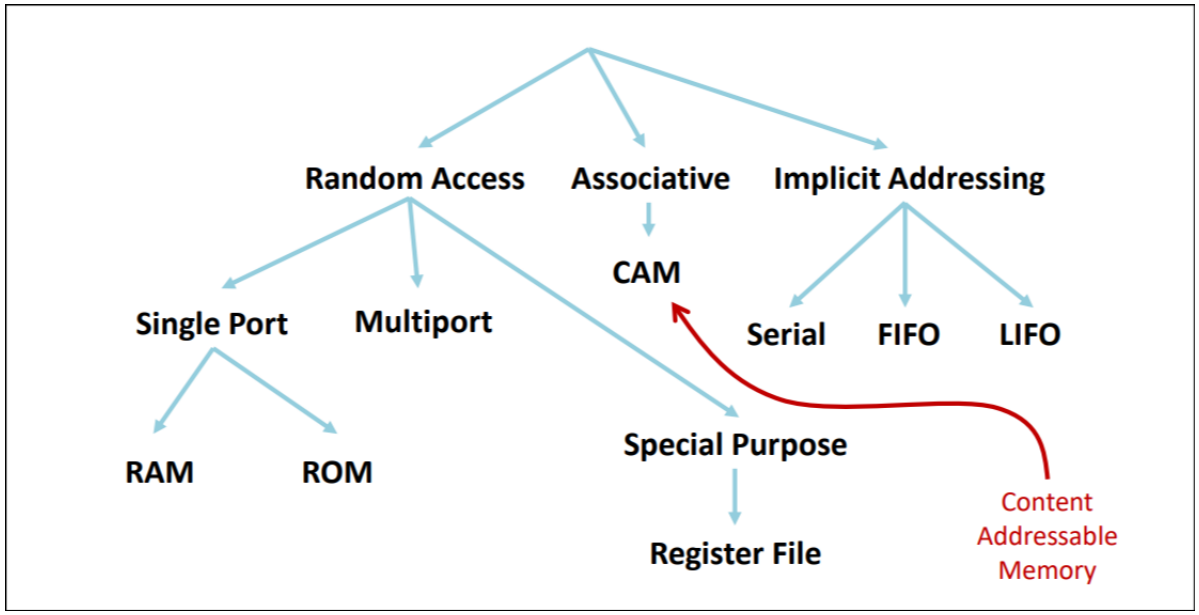
- Memory can be **very slow** $\implies$ Caches
- Memory is *finite* (relatively small) $\implies$ Virtual memory
- Memory can make an **ISA too complex** $\implies$ pipelining

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

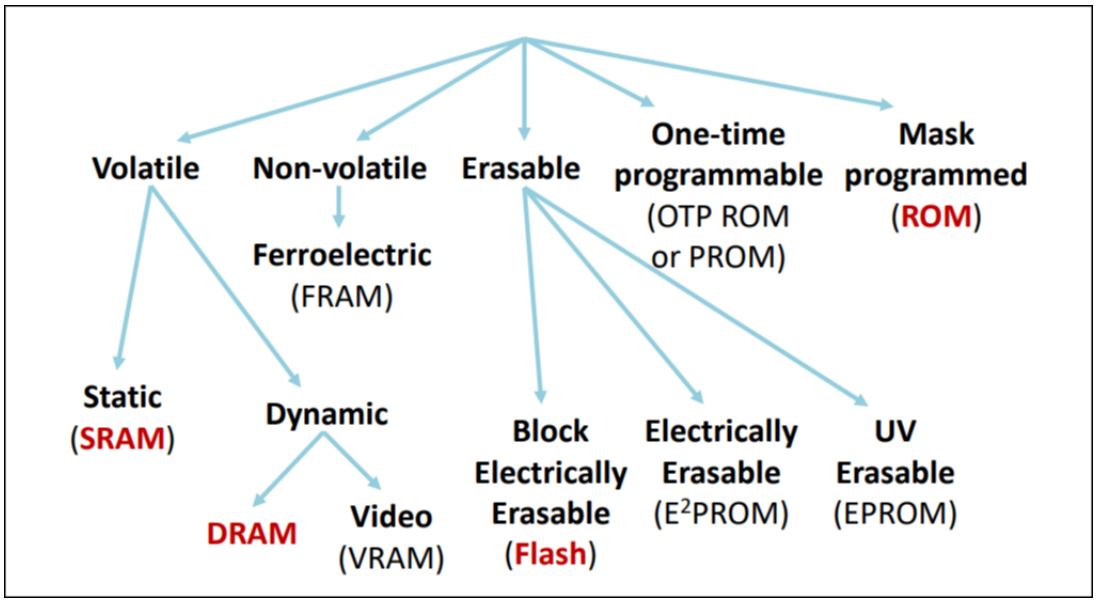
- SRAM, DRAM, EPROM, Flash, etc.

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

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

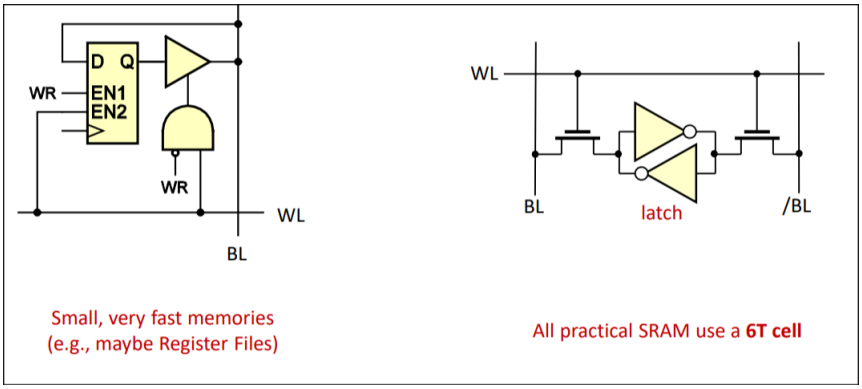


We have here all the type of memory that can be used, What we use when we program are Random access memory, this means that the memory can be accessed with an address. In the tree of random access memory we have:



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

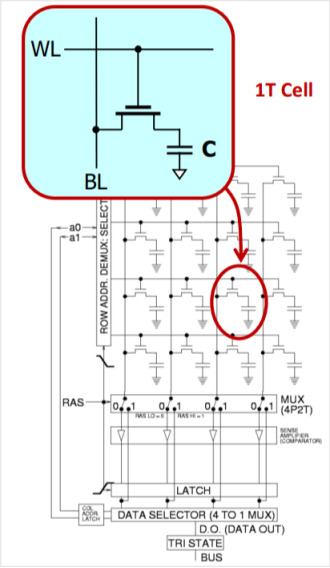
it is stored for infinity.
The only way to write is to "*shout louder than the current signal*". Imagine we currently have a 1 as the output of the latch and I want to put a 0. If I shout 0 louder than the 1 while connected, it will have a short circuit... and this is bad. **However** what is going on 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:

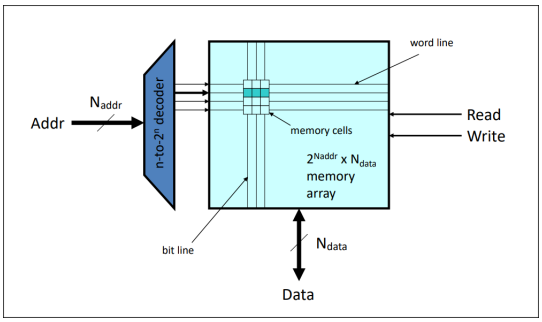
DRAM

- 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.

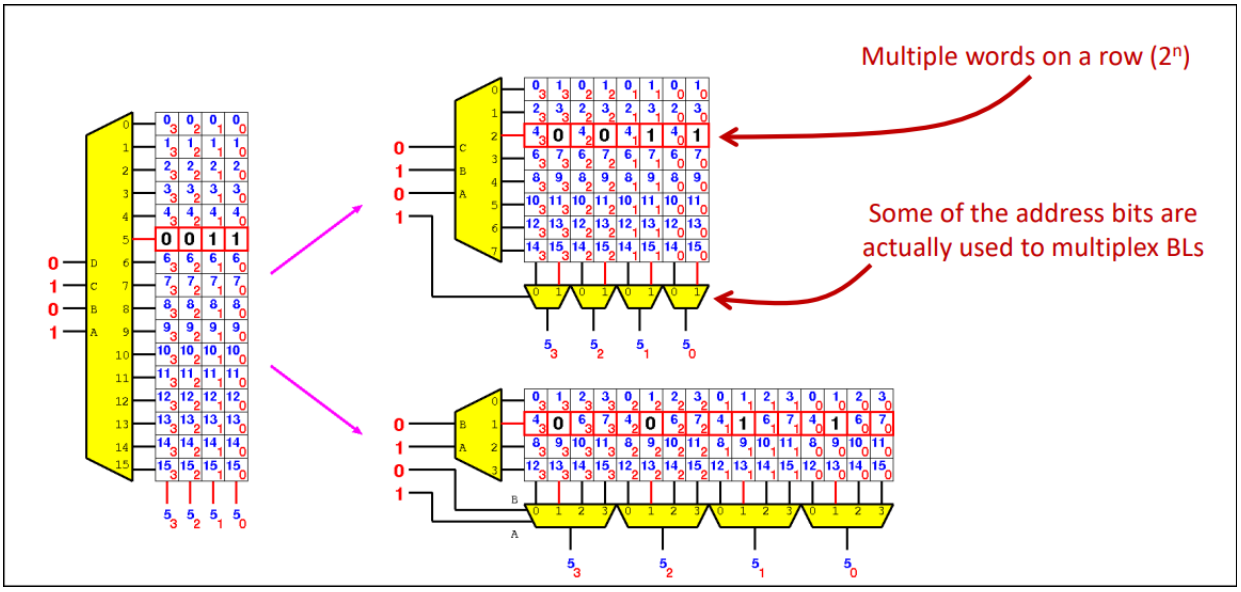
So imagine, if we don't go in each cell every 60ms then we lose the information, but we have other things to do? So how do we do it? - we have someone else refresh them for us. The memory controller is reponsible for refreshing the contents of the DRAM instead of the CPU.



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:



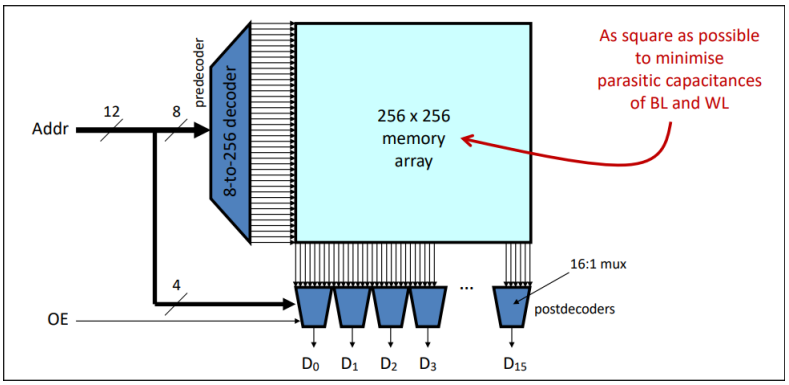
Here we have 3 ways to do so:

- 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.



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

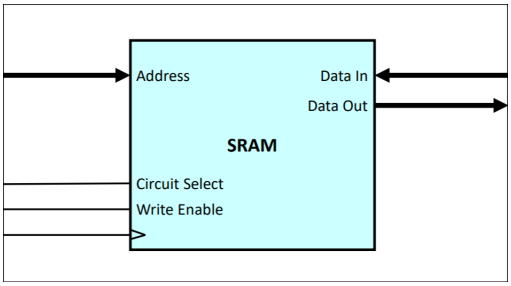
$$A = rc$$
$$\frac{A}{c} = r$$

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.

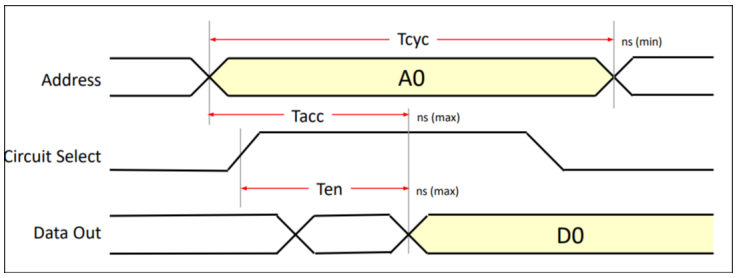
Static RAM typical interface This is the typical synchronous SRAM that we have already seen before:



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

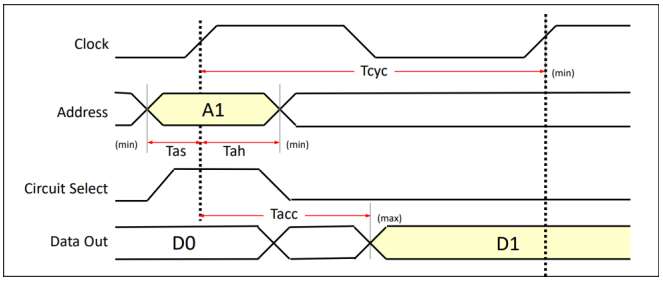
- Enable the memory → assert the address → wait for the data
  - Data out is available after a combinational delay T_{acc} = Access Time
- Maximum frequency is limited by the minimum T_{cyc} (time for a cycle, time for us to be able to change the address)



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

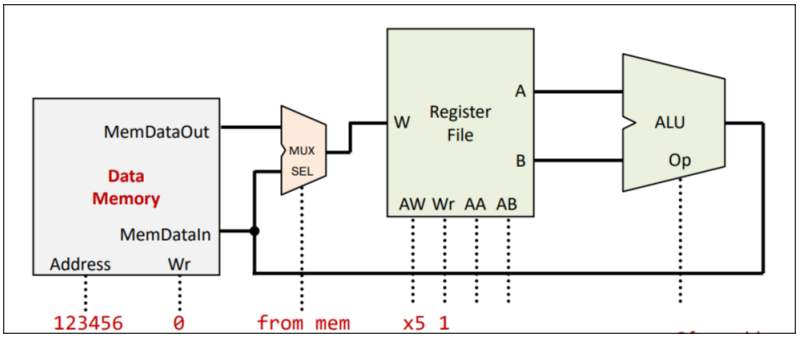


Load and store instrucitons

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:

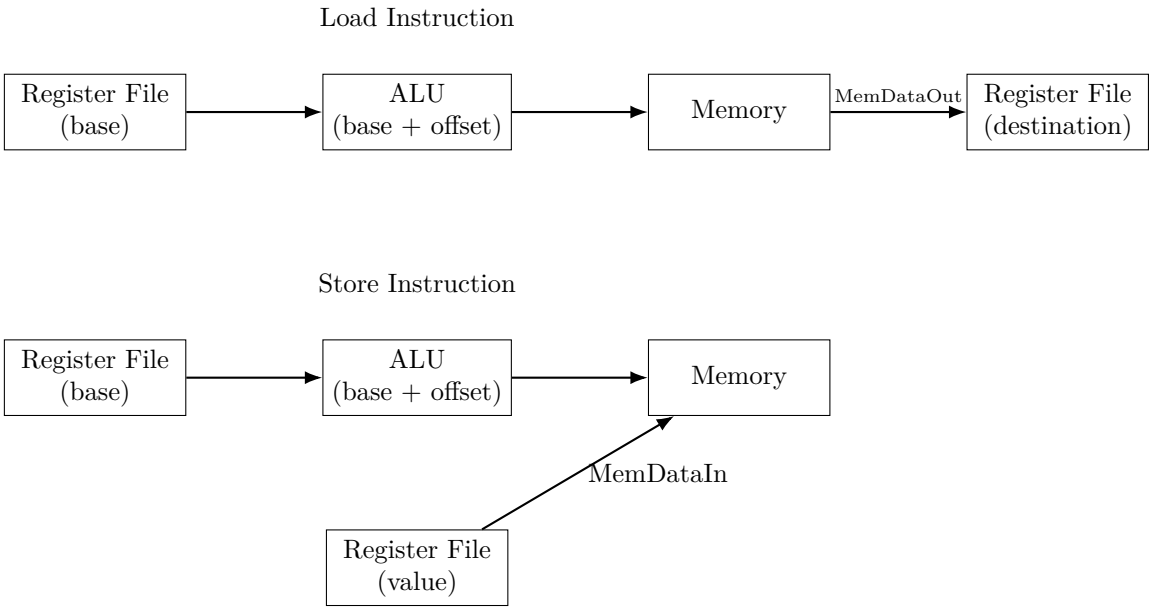
```
lw x5, (123456)
```

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:



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?

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]
```

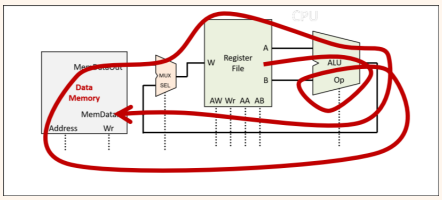
However those instruciton **does not exist** in RISC-V.
RISC-V is designed to have two world: one for accessing memory, and one for the logic/arithmetic etc. We cannot mix them together.
However in x86/x64 we can do this:

```
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:

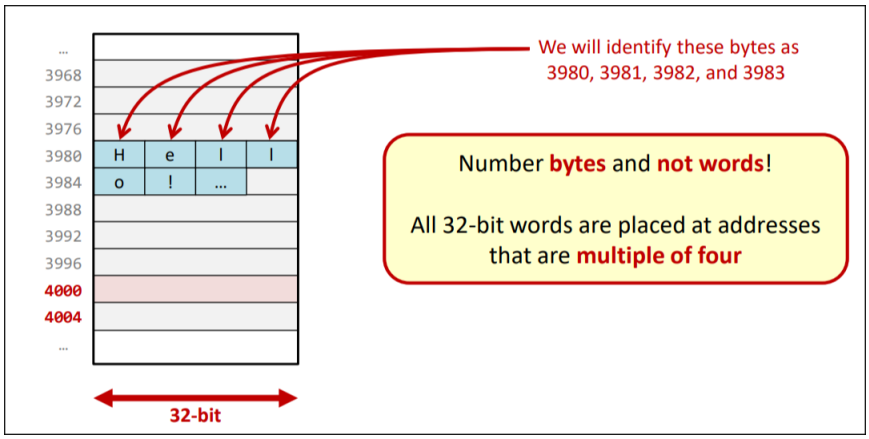


We would need to go like four times to the ALU, the fact is that **this is possible** however it makes it hard to optimize the processor.

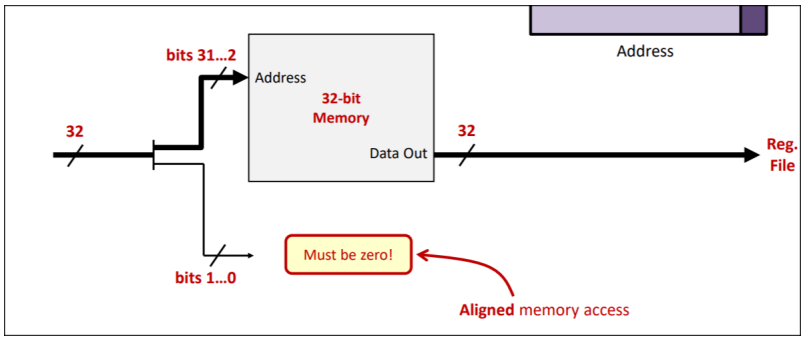
The question behind all this is how does intel can still be as famous as they are now with instruction that looks like this and that cannot really be optimized? There is still the majority of the processor to this day that are intel processor (even if amd is better...), we will see this in a couples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

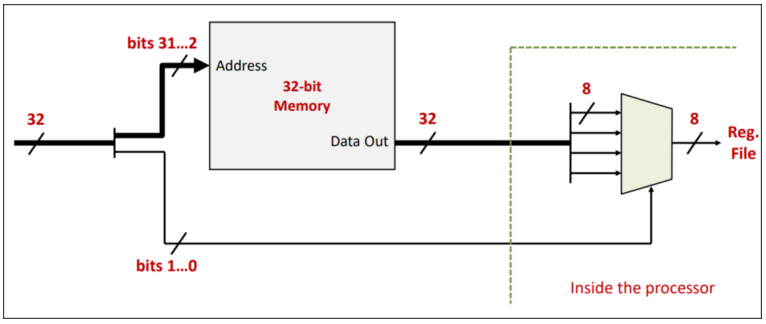


Loading a word (`lw`) and Instruction For instance, if we are interested in word, we cannot look for the word at address 3981, this would'nt be a word.



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.

2.4 Arrays and data structures

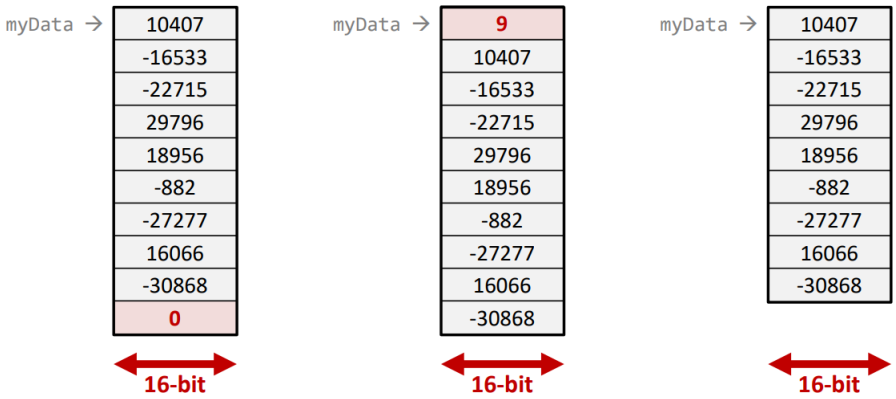
Data structures is one of the main concept in computer science, even in this course we have already talk about it (the stack).

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).

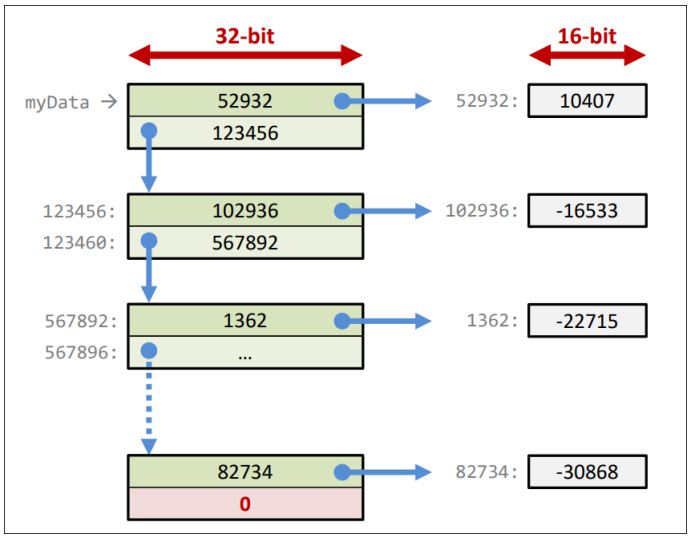
```
addi_positive:
        li t0, 0 #t0 will hold the sum (initialized to 0)
        mv t1, 0 # t1 will hold the array index

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

2.5 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)

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

- 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**

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

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

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

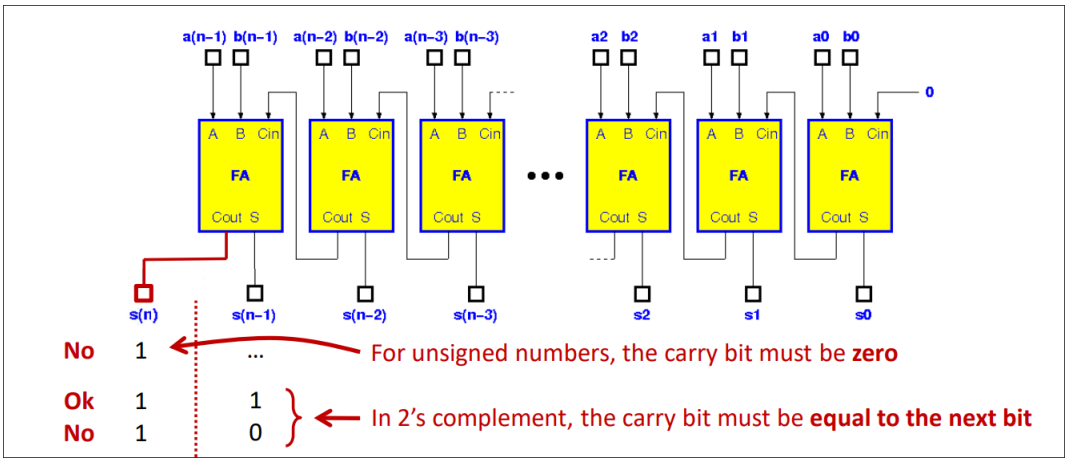
For the next part of this course I am gonna skips this because it is a big review of what we have already seen in fds, I am just going to write what I find intresting **for me** which is not necessarly the most important thing.

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

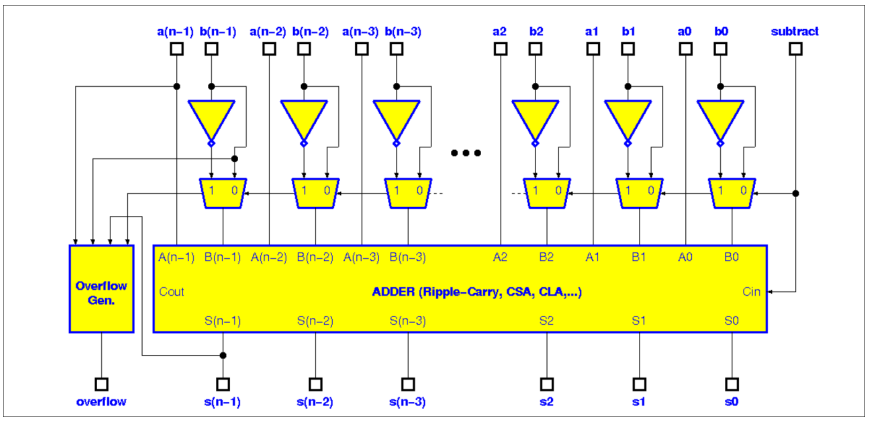
$$-A = \overline{A} + 1$$

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 $\neg b_i$



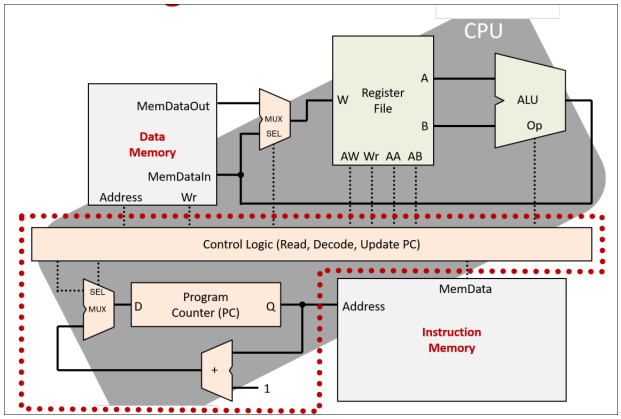
as we can see this allows us to put the substraction and addition into the same module.

Chapter 3

Processors, I/Os, and Exceptions

3.1 2a. Multicycle Processor

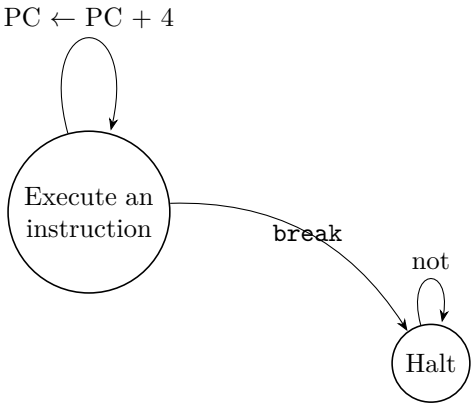
In this section, we will more go into the detail of the **hardware** behind the cpu. Especialy multicycle processor.
As seen before, the CPU has more than one part:



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:

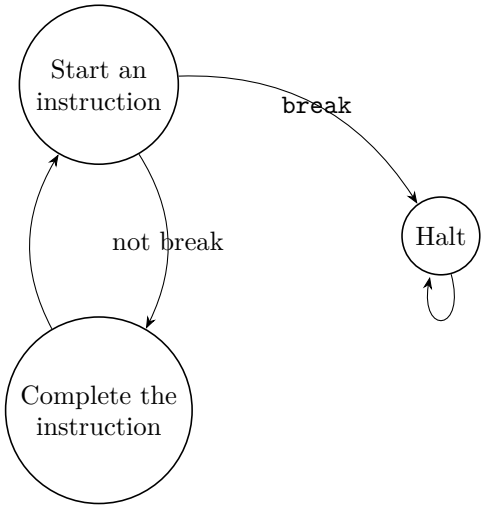


Remark If someone knows how to makes the break here above or below the line I will be curious how.

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:

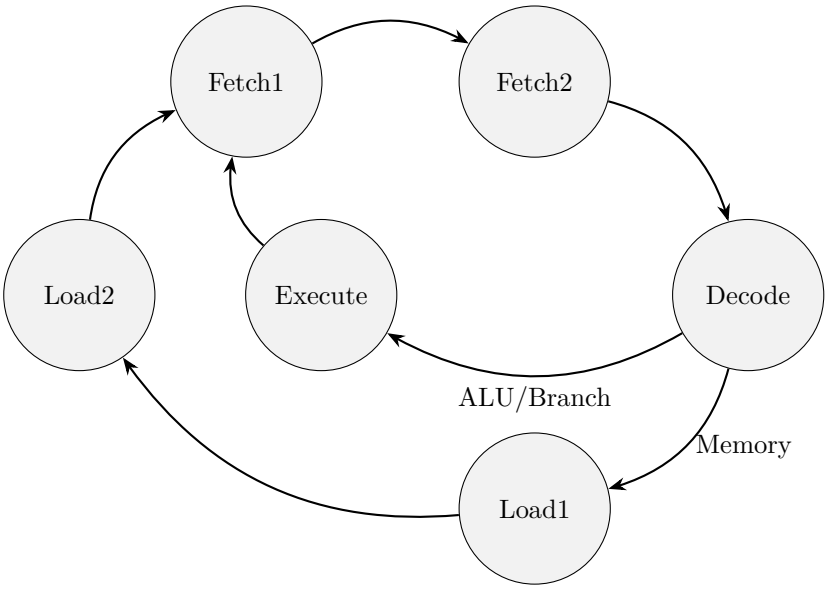


The question we must ask now is: Did we gain anything?
At the moment not really, before we had 1 instruction **per cycle** at frequency F. Now, we have 1 instruction every **two cycles** at frequency $2F$.

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.

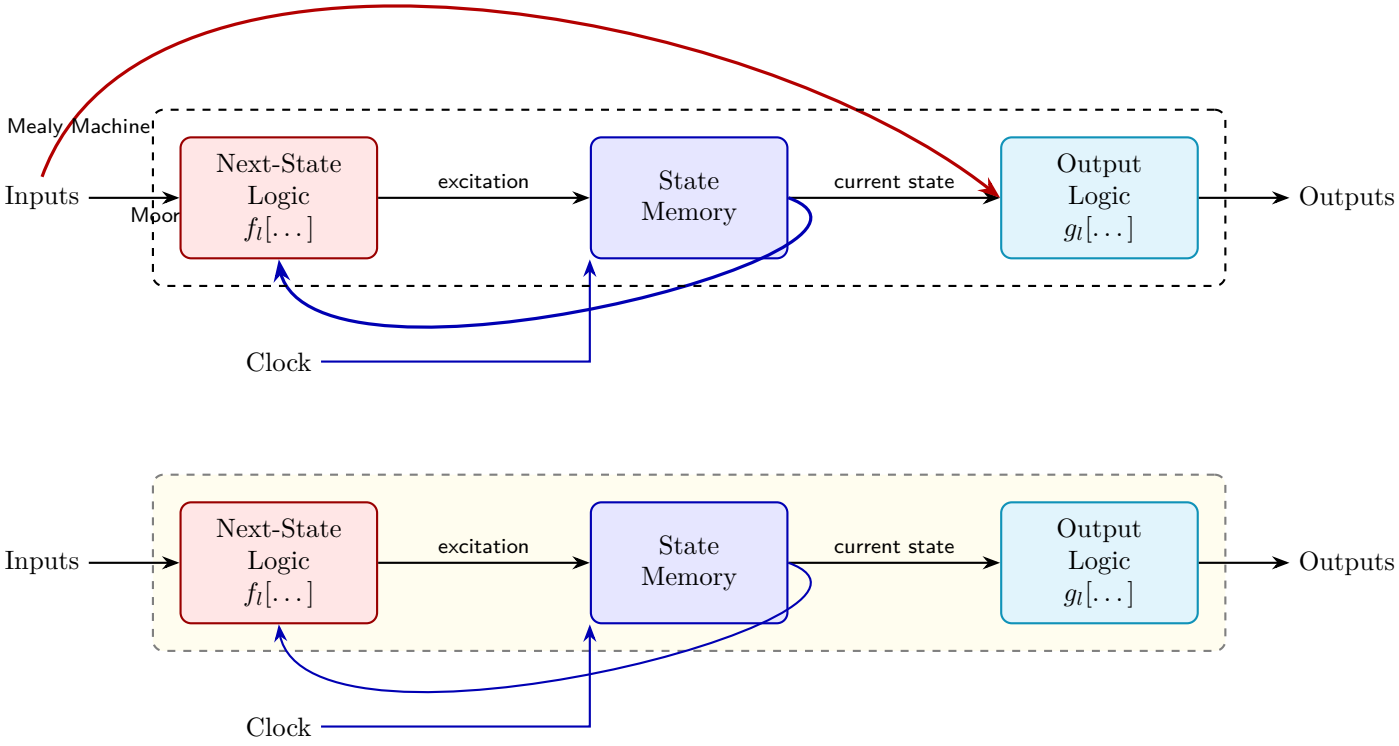
Multi cycle Processor



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**.

3.1.1 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:

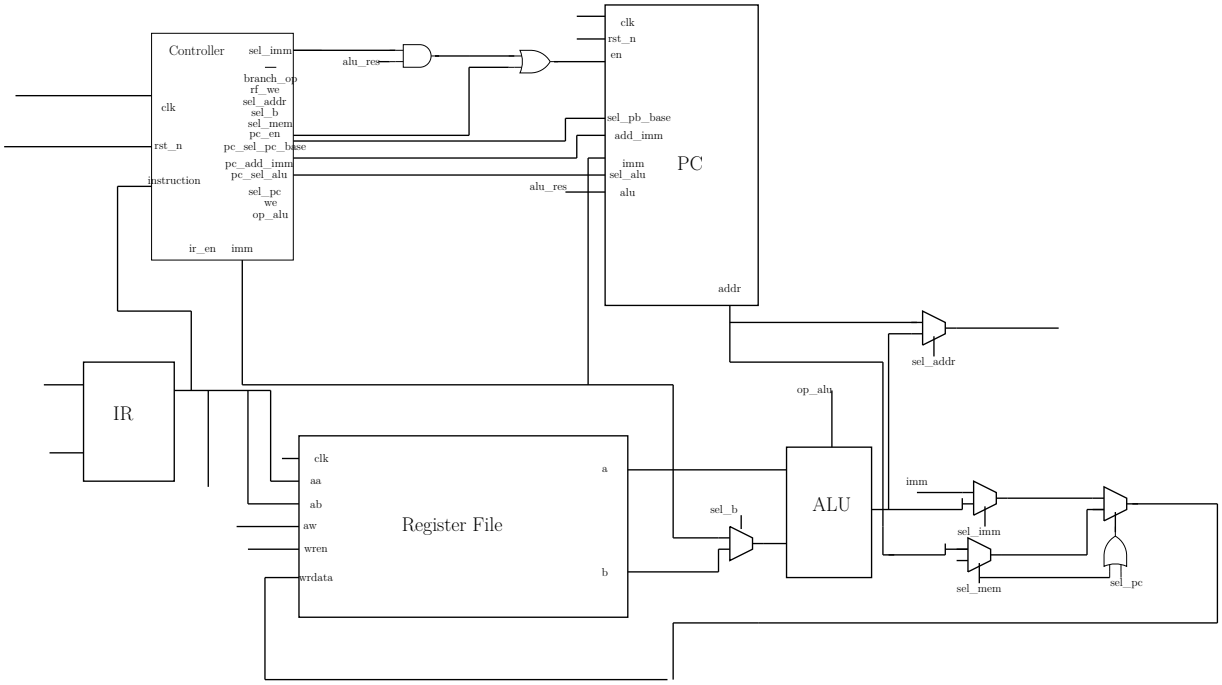
1. `branch_op` $\rightarrow$ informs us of if we are in a branch operation
2. `alu_res` which is the lsb of the `ALU` as said before

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

> 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

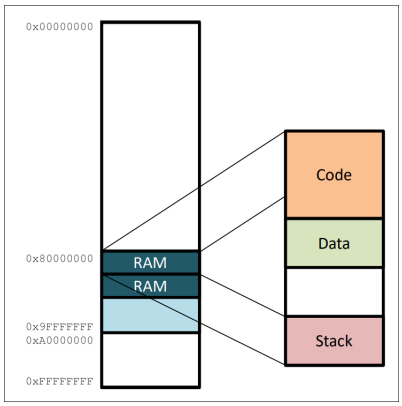
`sel_pc` signal.



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).



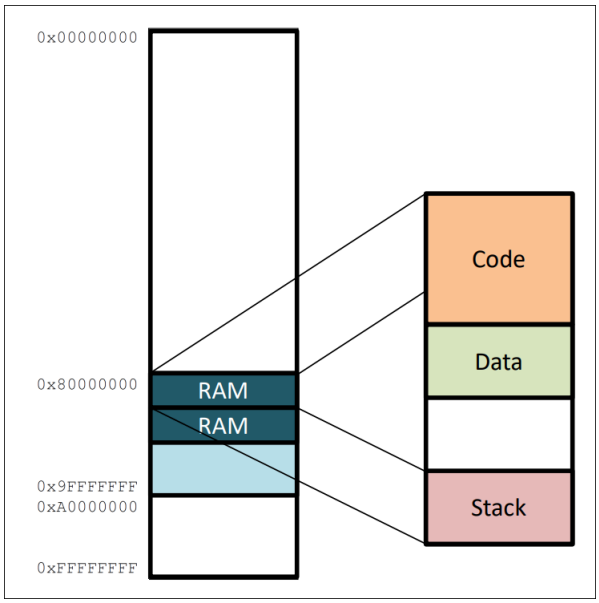
3.2 2b. Processor, Inputs and Outputs

The cpu

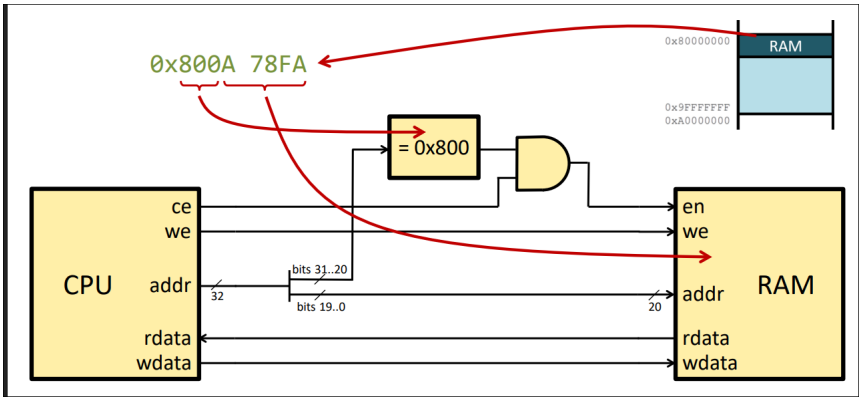
As said before, the cpu is a very **sequential** component (but from now on we may omit the **clock** in diagrams). Therefore, here we only have one data bus for read and write in our processor. The question we have is how can we handle input and ouput with only one data bus?

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:

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 3.1: Approximate data rates for common peripheral interfaces.

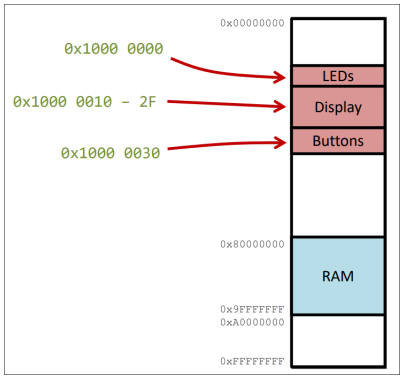
Accessing I/Os: Port Mapped I/O (PMIO)

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.

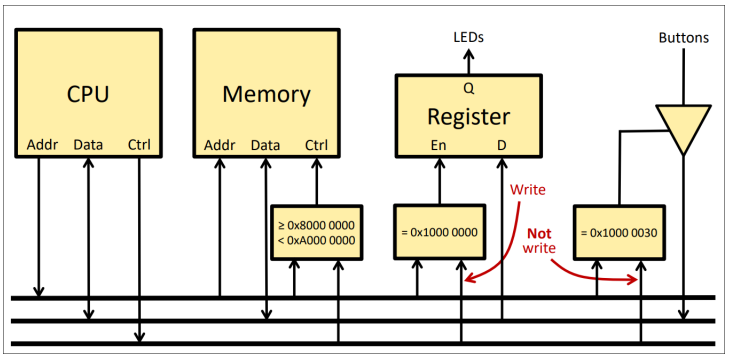


This means that there are no special hardware needed in the cpu $\implies$ no special instructions needed. For intsance in our example if we wanted to write a new value in the led:

```
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:



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.

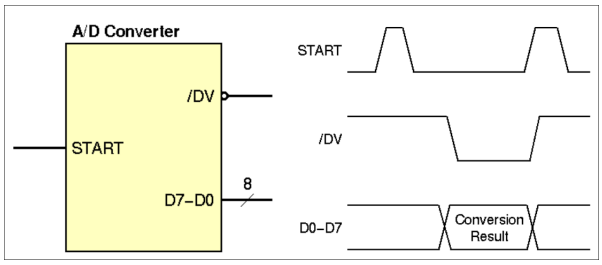
- **PMIO** separates memory and I/O with dedicated instructions and control signals, providing clarity but requiring extra CPU support.
- **MMIO** treats I/O devices as memory, simplifying the CPU design and instruction set at the cost of shared address space.

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

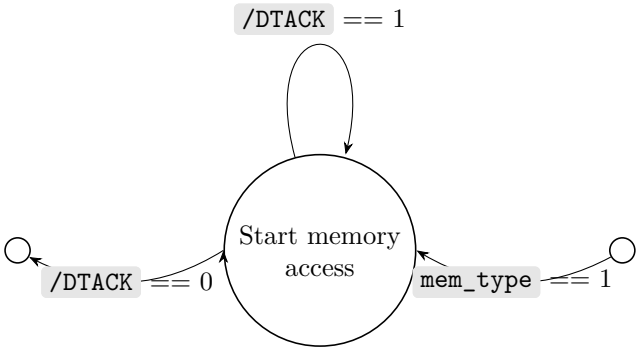
This is similar but not identical to the MC6800

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.

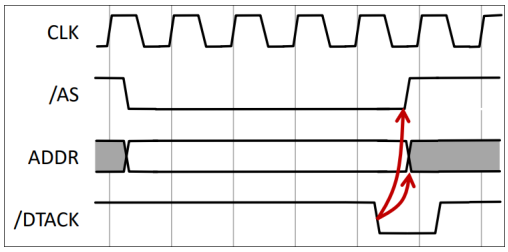
The goal now for us is to create a circuit that is able to connect the A/D converter to the processor (the MC68000). For this we will use a **memory mapped interface**.
We want:

- **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**.



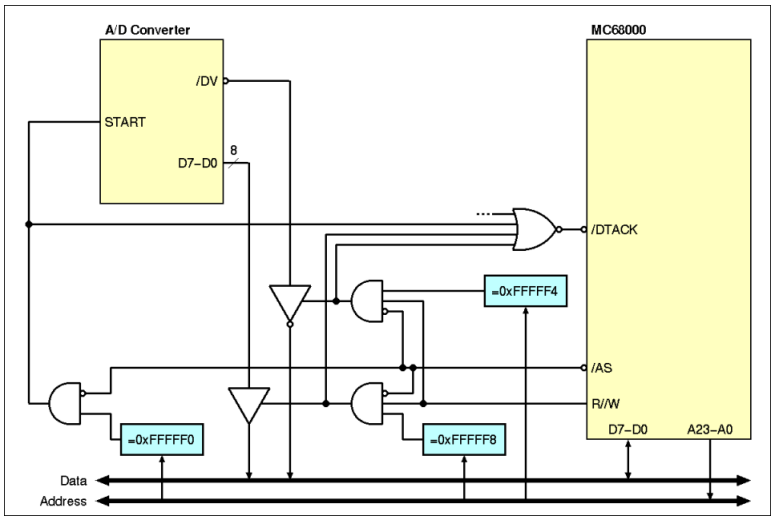


Figure 3.1: A/D converter circuit

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

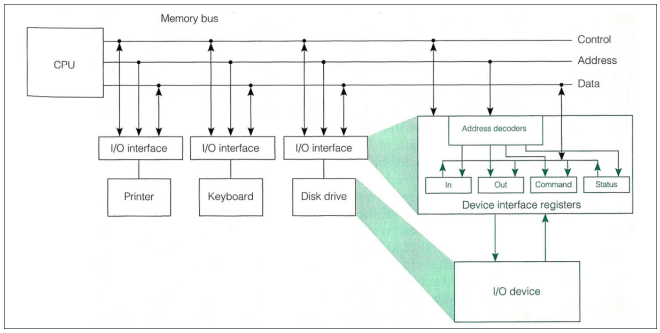
Let us now loot at it as a software person which is pretty easy because of the MMIO:

```
read_adc:
```

```
        lui t1, 0xfff
        addi t1, t1, 0xff0 #t1 = 0xffffff0
        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**.



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...

3.2.1 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

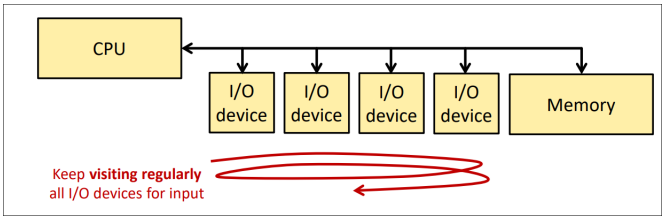
For instance we can try to send a String into the UART:
Just to remember, a `String` is an array of char which is terminated by the null character.
The goal here will be to send each char (bytes) to the data bus. To do so we'll

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.



3.3 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**

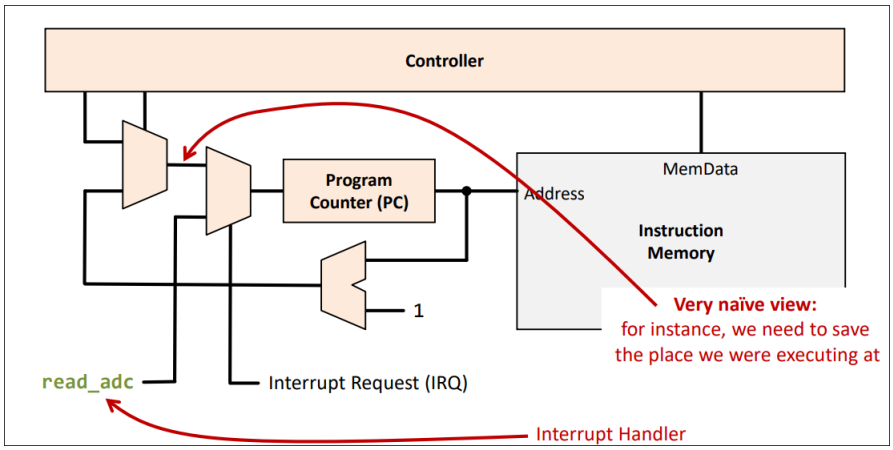
Seen this already in some languages

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

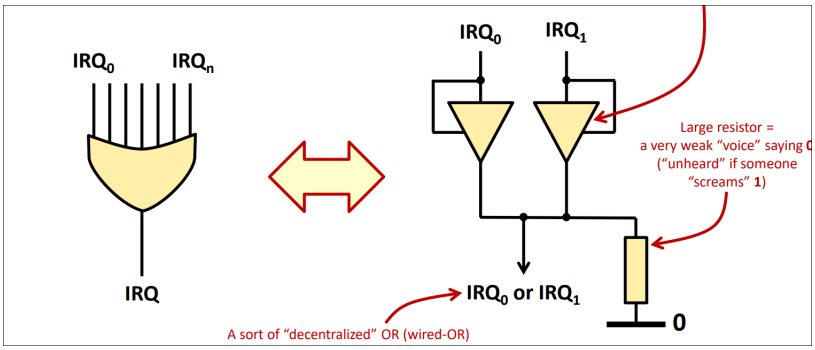


We use the `read_adc` address which is where we handle the I/O interrupts. The interrupt request serves to trigger the interruption.
However there are several issues to take care of and behaviours to define:

- 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

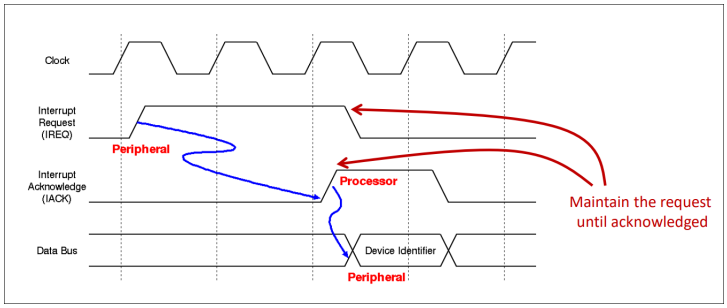


Figure 3.2: tiaming diagram

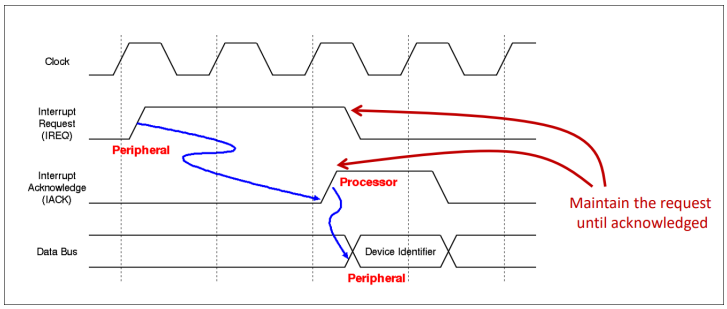
On the software side, it works with the same code we used before for polling. However, the `read_adc` function is now called by the interrupt handler. The interrupt handler stores the address of the program that manages all interrupt requests, and from there, it launches the program that handles the ADC.

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:

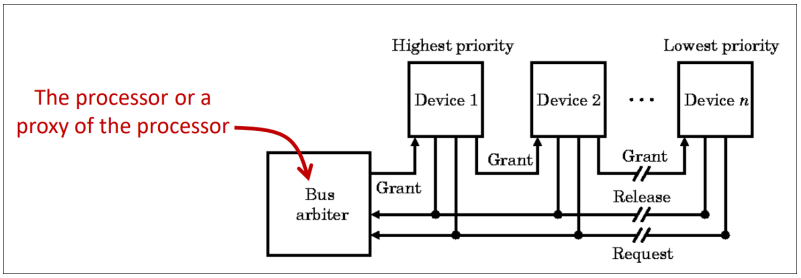
- Anyone places request (IREQ, Request)
- Acknowledge line (IACK, Grant) passed from one device to the next
- Device which wants access, intercepts the signal and hides it from succesisce devices
- Simple but (1) slow and (2) hard priorities (meaning hard in like hardcode something).



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.



3.3.1 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.

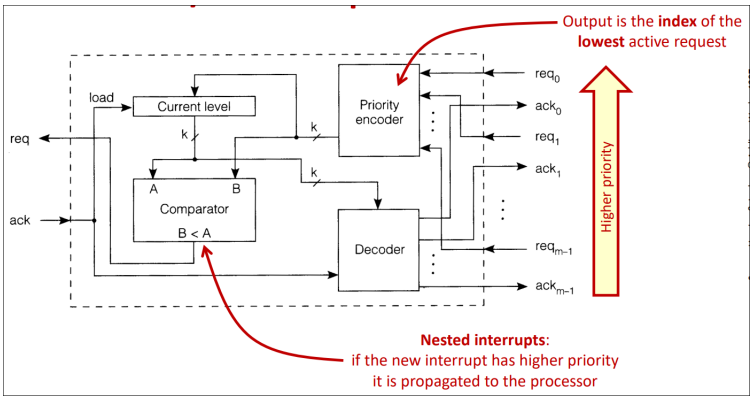
With the DMA

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.



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

Définition 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

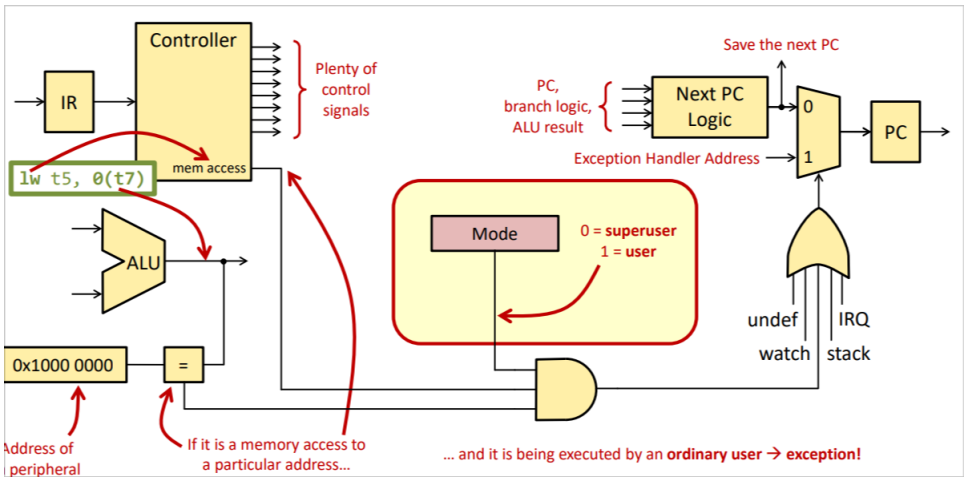
An example sequence of operations with a timer and DMA could be:

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.

3.4 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

> 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**!

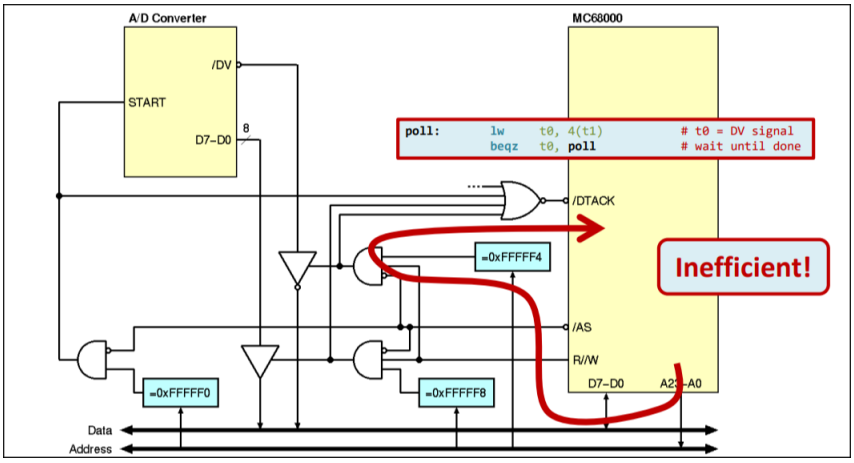
- Maybe the **stack cannot be used** (e.g., the exception results from a stack overflow)
- Maybe the **exception handler cannot be interrupted** (e.g., the handler uses static locations to save data including `mscratch`) and is therefore a nonreentrannt procedure)
- Maybe the **system cannot withstand 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.

> 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)

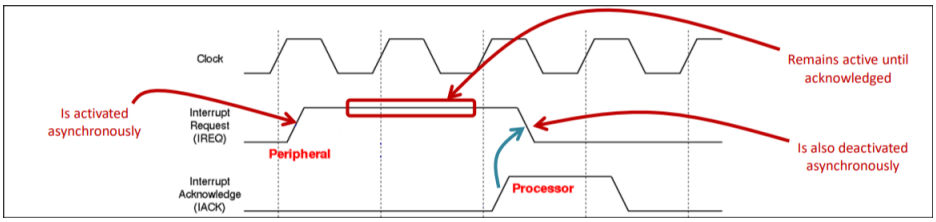


However this is pretty ineficient, the processor has to always check weither the signal is active or not instead of doing real work.
The goal here is to improve the interface to the A/D converter so that:

- 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`

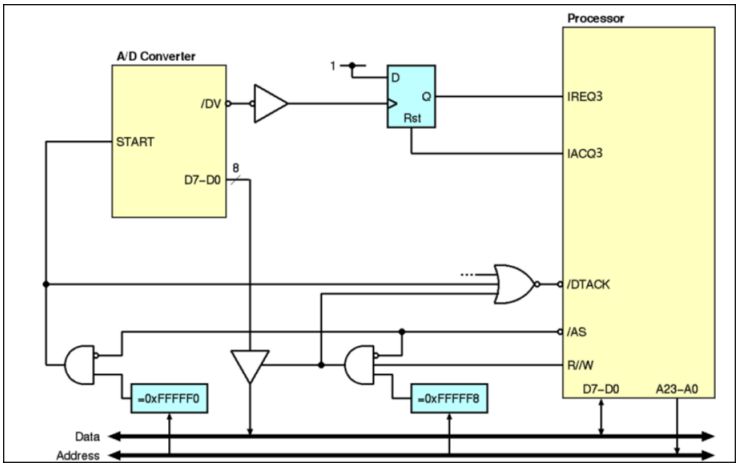
Suppose that our 8-bite preoccessor has an internal interrupt controller with various `IREQ` / `IACK` signap pairs for I/O interrupt requets
We have been assigned for our ADC these:

- `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 = 0x
                            fffff0
                        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
```

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 02
    d
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
```

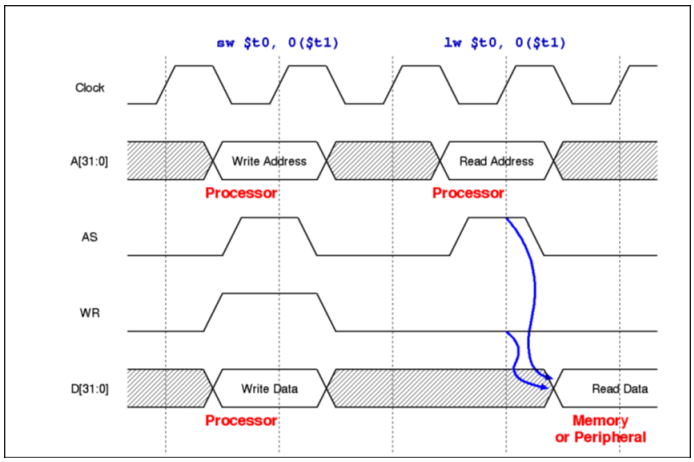
lectureDate2025-10-17 . lectureDate — **Cours 12 : Example of I/O/s and Exceptions** Today's lecture is an example of what we did the previous weeks.

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.

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

Chapter 4

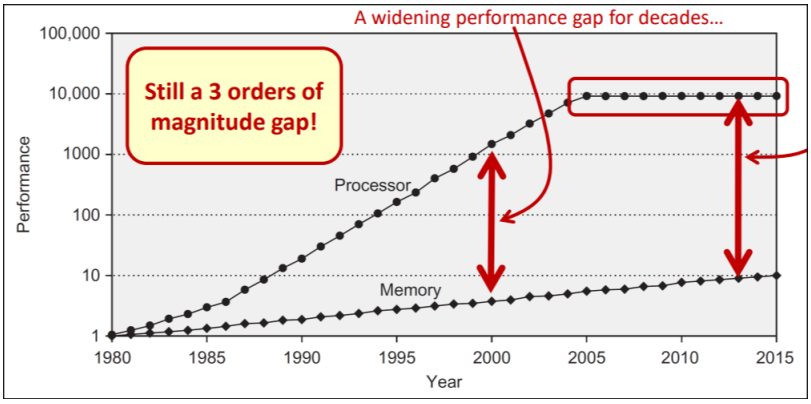
Memory Hierarchy

lectureDate2025-10-17 lectureDate — **Cours 12 : cache**

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.

4.1 Caches

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



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.

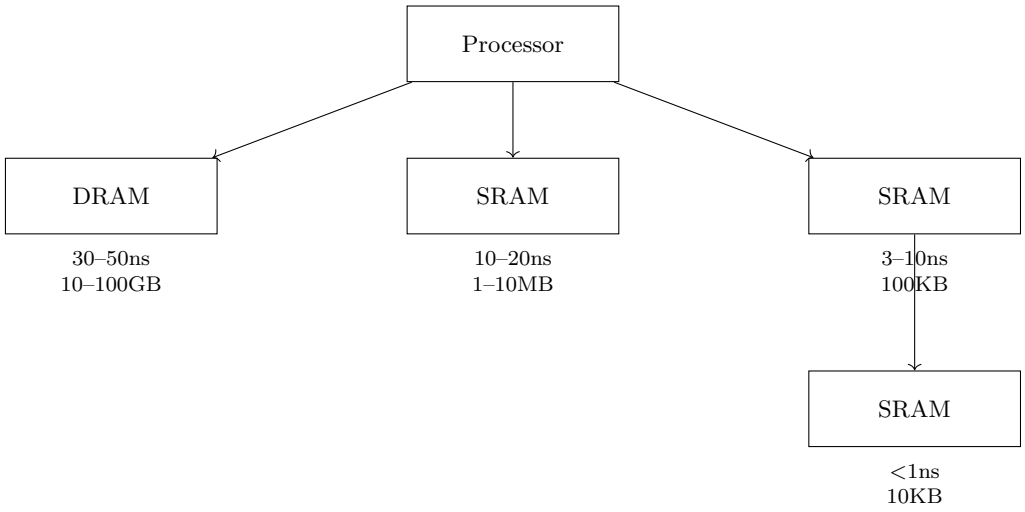
Processor has improved a **lot** in comparisaison of memory.

Remark When we say *memory*, we mean `DRAM`.

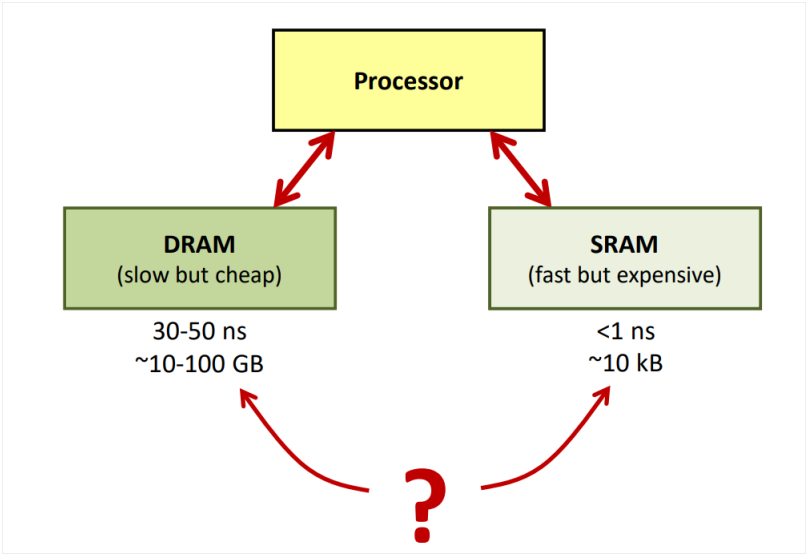
The issue is that we want a **big** memory and a **fast** processor which cannot really work well together, we need to **do something** ourself.

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.



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

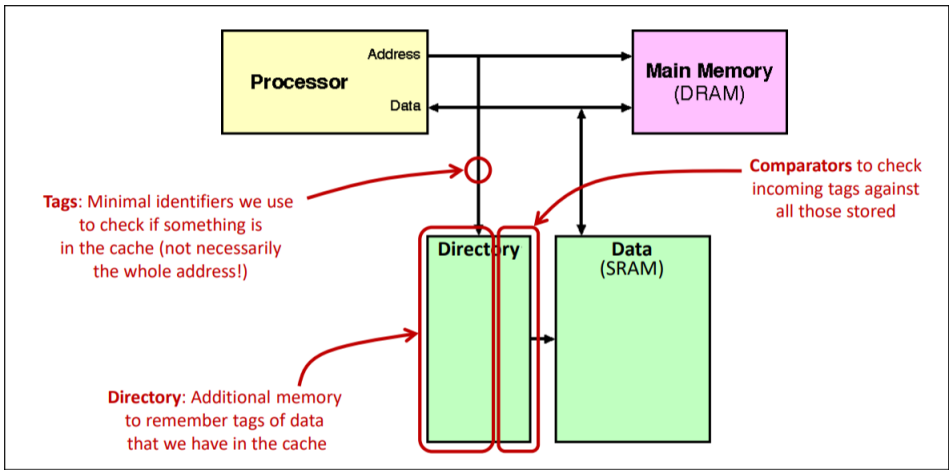
- One could analyse data structures and program semantics to detect easily used variables/arrays and thus decide placment → OK in some context (emebedded) but we want to have the programmers not to go through this hassle
- To do so, we will add **hardware** to help

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

Définition 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**.

Définition 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

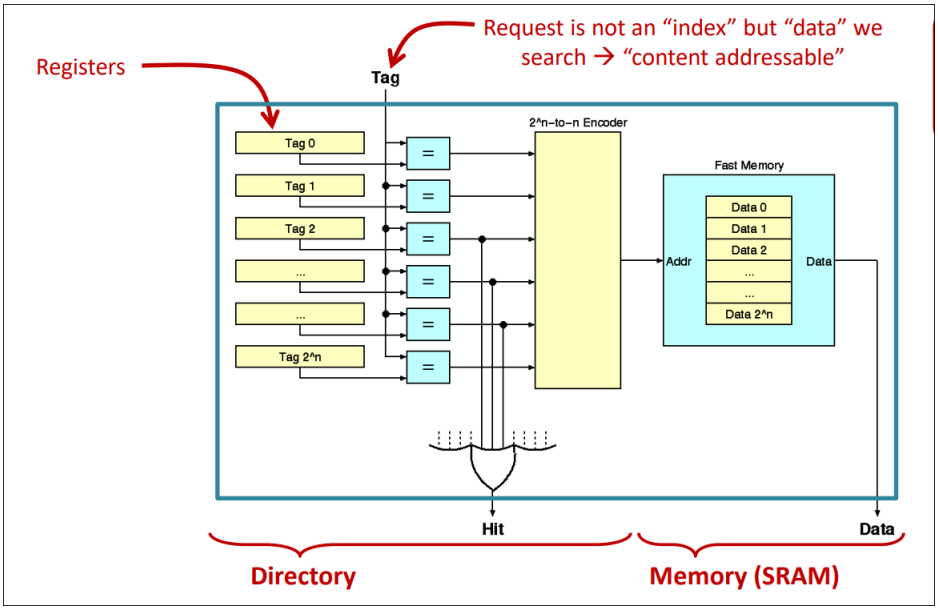
And as we can see we have the `Tag` which are the content and the address which is the output of the encoder.

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



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 :

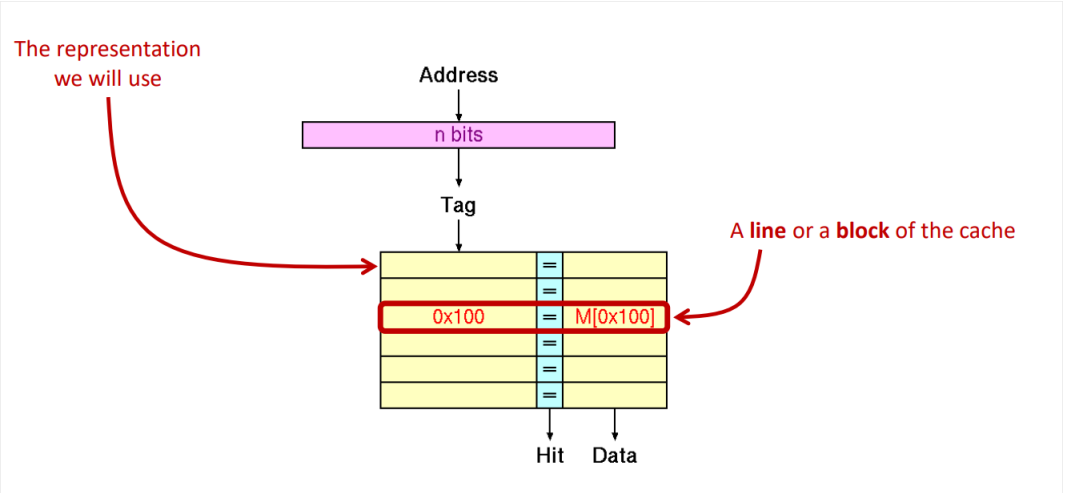
- **Direct-mapped cache** → each memory block maps to **exactly one** cache line.
- **Set-associative cache** → each memory block maps to a **set of cache lines**, and can go into **any line** of them
- **Fully-Associative cache** → a memory block can go **anywhere** in the cache.

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)).

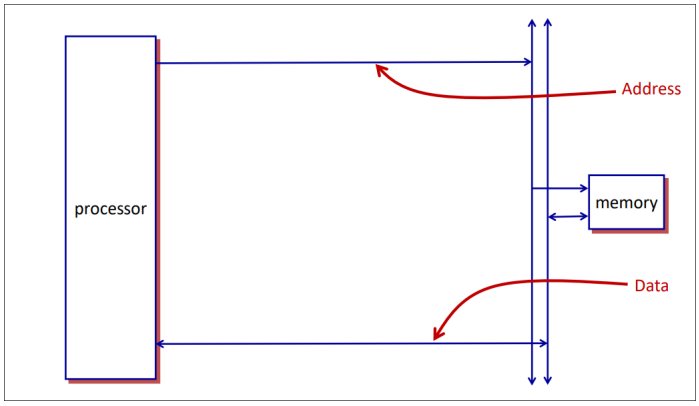
The issue here also is that **all those comparators** are pretty expensive, for each tag we have a to have a comparator which is pretty costly.

The representation

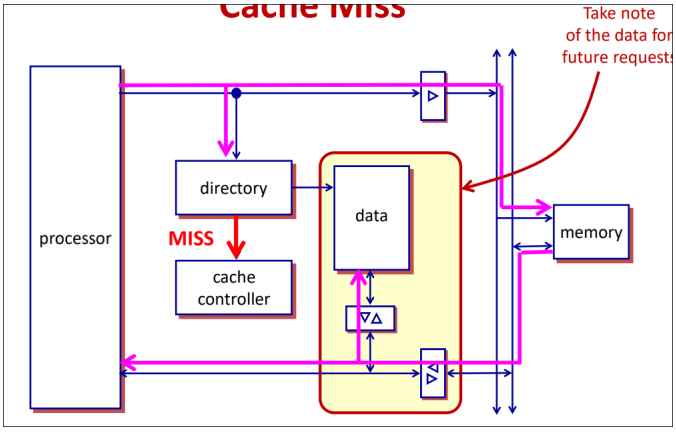


So this diagram is **the same** as the one above we just simpified to make it easier to *understand*.

Cache and Cache controller

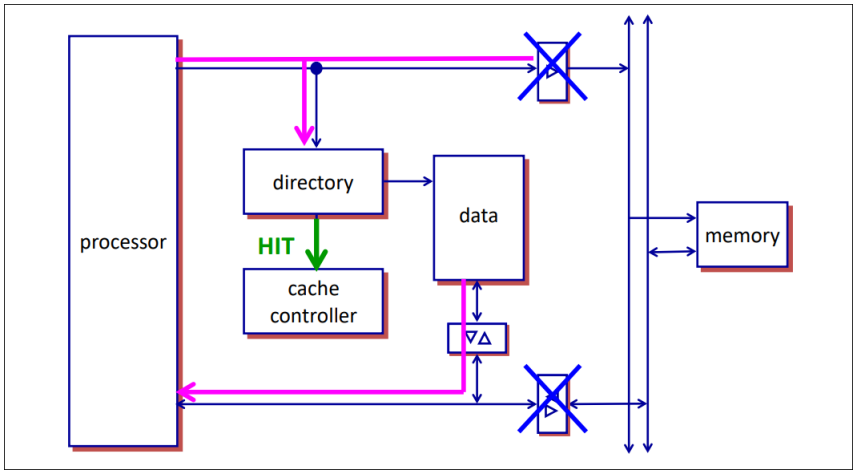


The difference from before is now that the processor **talks to the cache** and if the caches doesn't find the content it send the address into the memory:



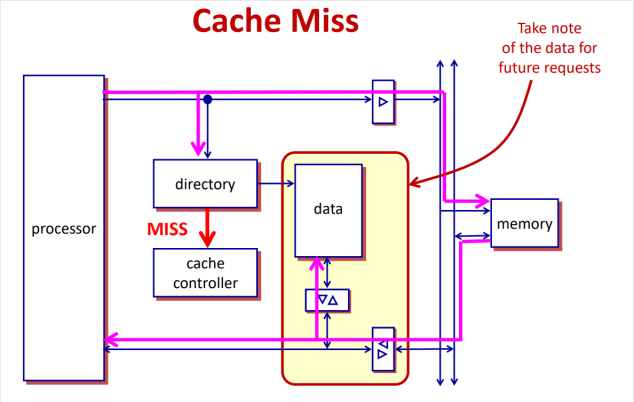
The main character here is the **cache controller**. It is the one that blocks or not the arrows (the little blue square) that let us cut or not the memory of the data bus. The cache controller here is a

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.

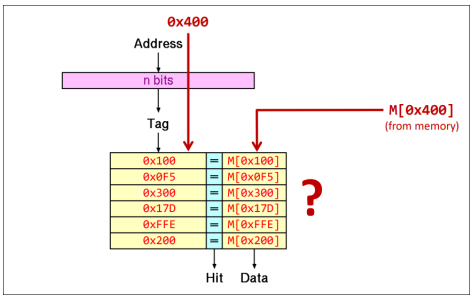


The cache is a hardware device

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)?

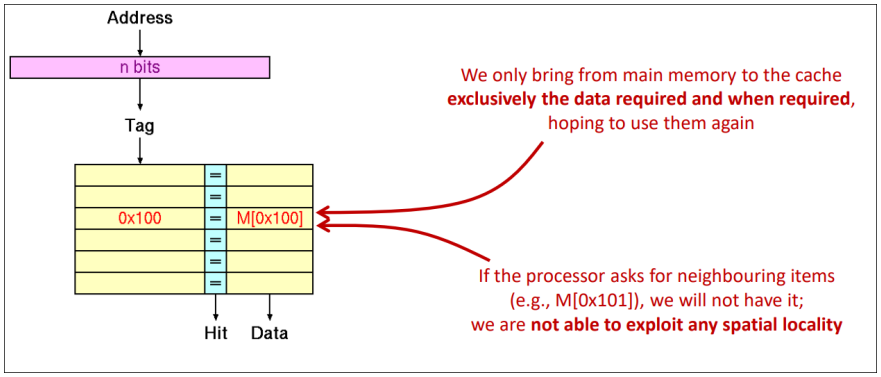
Eviction policy

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:

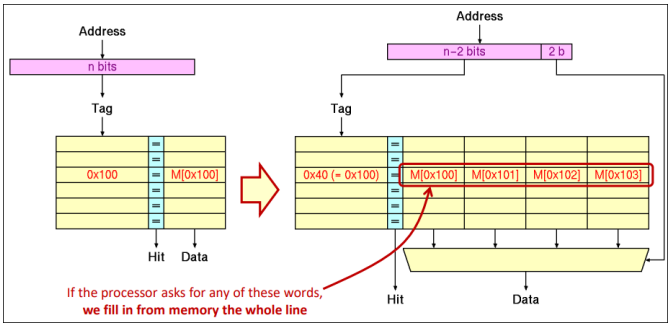
- **Least recently used** (LRU)
  - Replace the data that have been unused for the longest period of time
- **First in First out** (FIFO)
  - Replace the data that came in earliest
- **Random** → pick one at random and throw it away

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

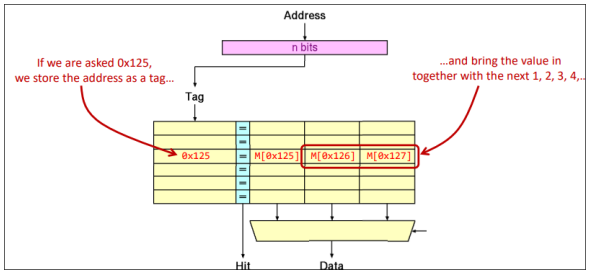
Only exploiting Temporal locality



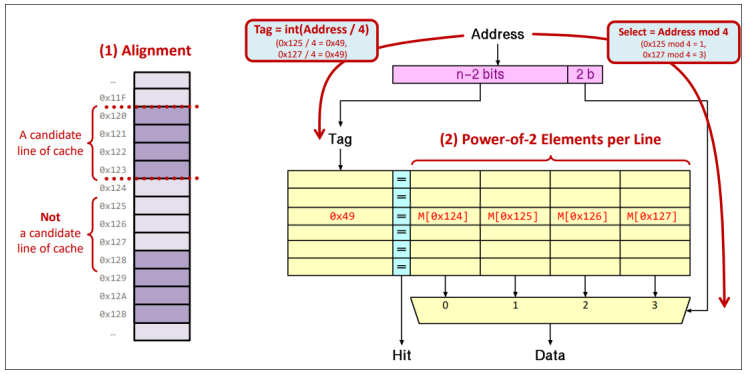
At the current moment we are only exploiting the temporal locality (the upper arrows that goes into data when we are fetching from memory).
A solution here is that when we are fetching the element at address `0x100` I also store the element at address `0x100`, `0x101`, `0x102`, `0x103`.



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:

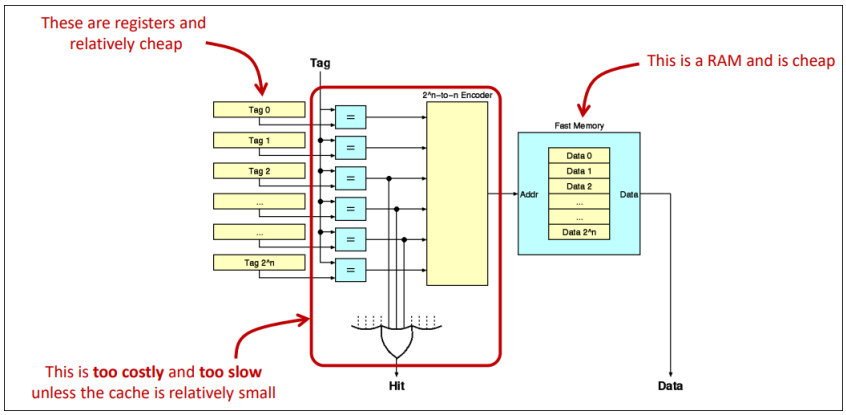


Here all our problems are gone! We only have to do a comparison and the data is easily accessed **and** it is **hardware friendly**. This means that this can be *easily* implementable.

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

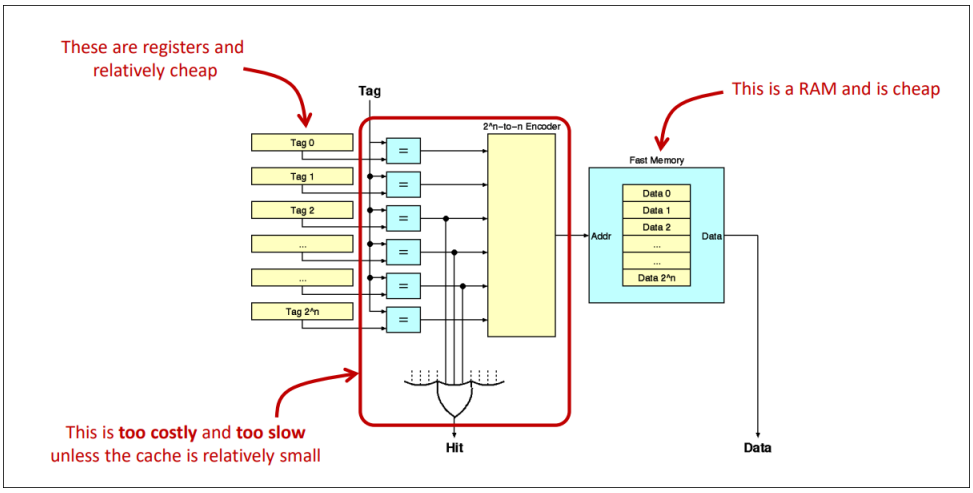
Okay all of this is pretty nice however, there is an issue that has still never been mention, how can we tell to the cache controller when there is a `hit` or a `miss`? The easiest way to do so is to put a big `OR` gate from all the tag. This is **slow**. Imagine having an `OR` gate with a million of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



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).

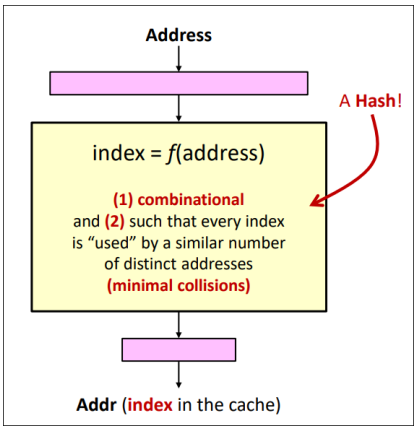
For this to works, every element should go into a **single place** of the `RAM`. I take the address from the processor, I figured the index in the cache, we'll get out **one** candidate word, we have the Tag here, we'll read and compare the tag and the content, if it is a `HIT` then we won else we `MISS`.

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`

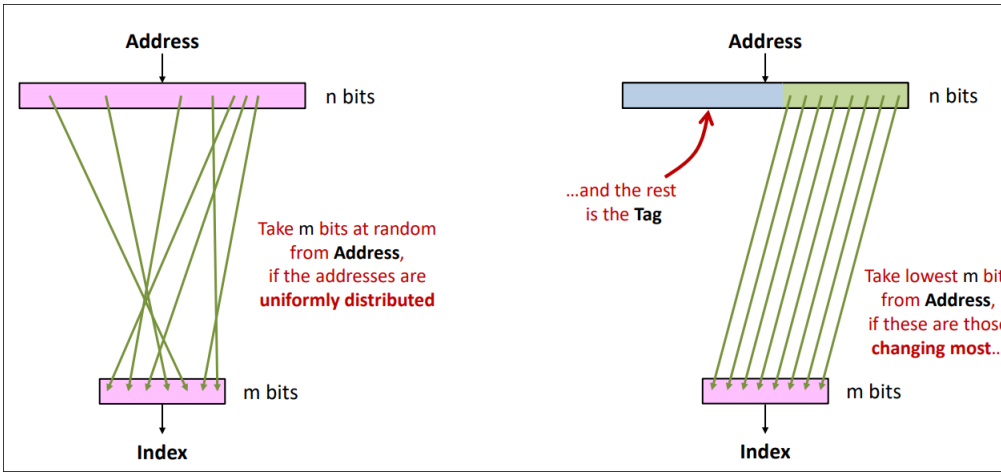
Now we are looking at the purple box above:



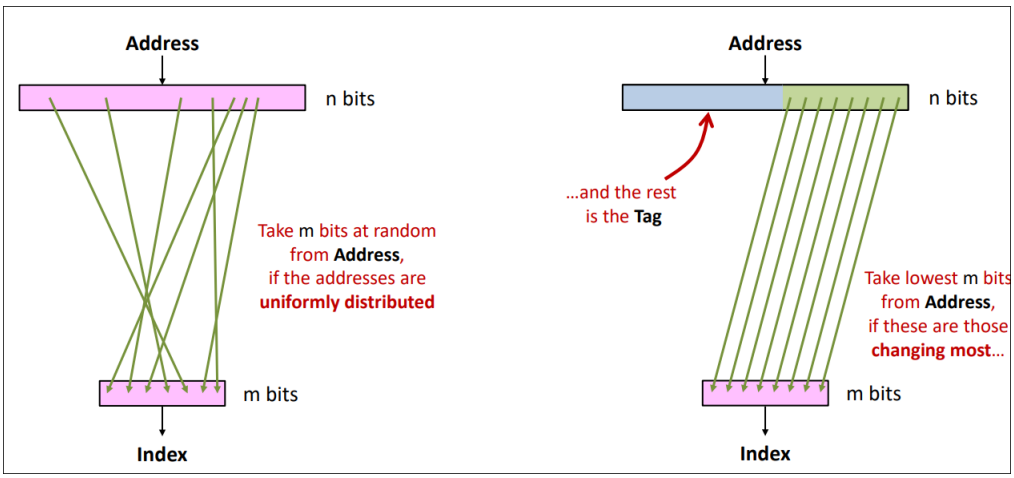
What we want to do is to go from the address in memory which is a 32 bits memory for instance, into a 10 bits memory which is the cache size. How can we do so? We are **Hashing**. We are taking less bits but produce the most different combination possible.

The simplest hashes

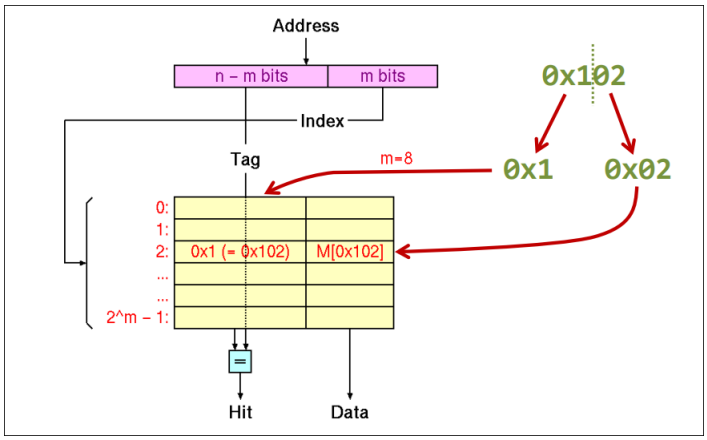
What we want as a hash here is a hash that is the simplest and the **fastest**.
For instance, we can just take some of the bits and we put them in a random (or not) order:



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** → 256 words per cache.
The question is how good are they, the criteria for this will be the number of hits.

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).

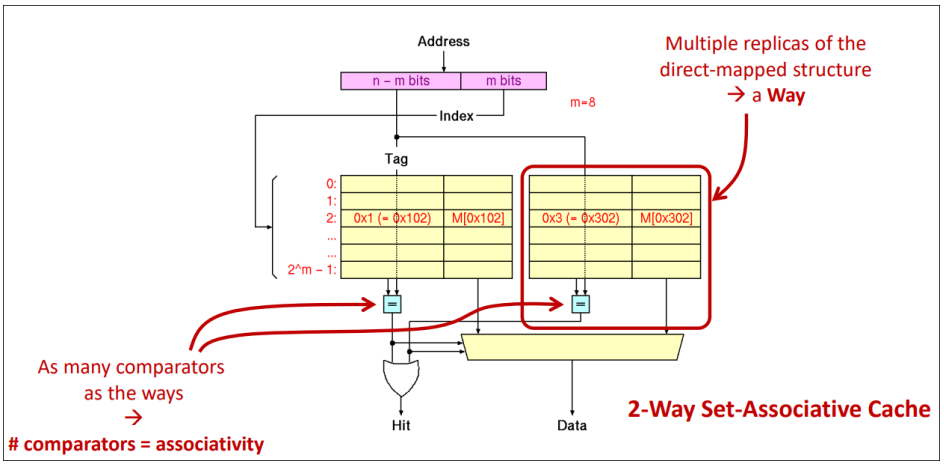
But what we can see here is that sometime the direct mapped cache can be very very inefficient. Is there a way to find the best out of the two worls: The speed of the direct mapped and the hit rate of the fully-associative one?

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

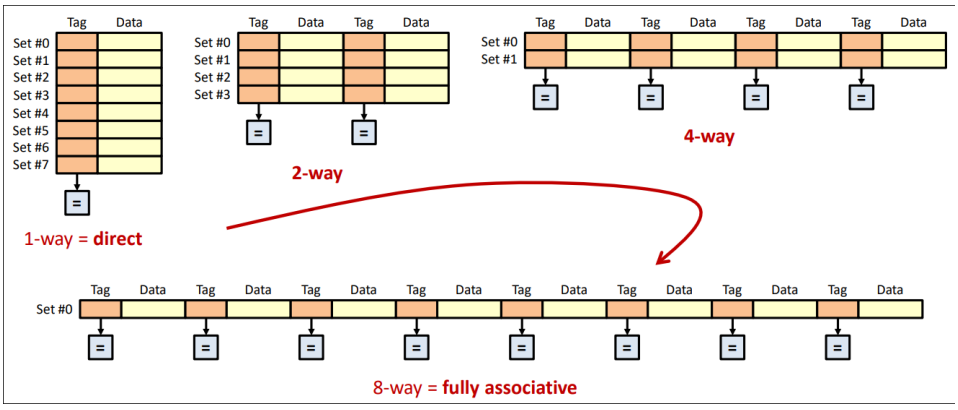


So the idea is to add a second cache map with the exact same direct-mapped structure. By doing this we allow a **second** element for the same index

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:



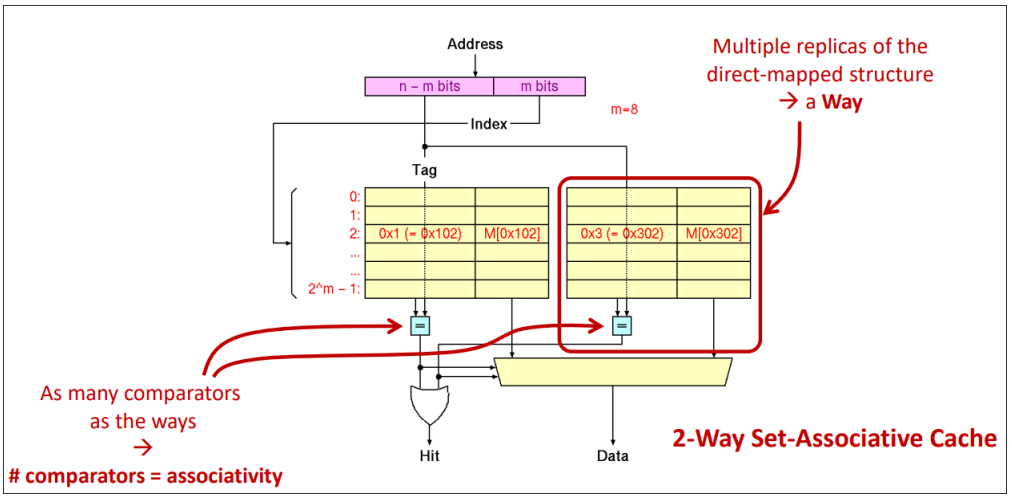
professor's question during the lecture

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?

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

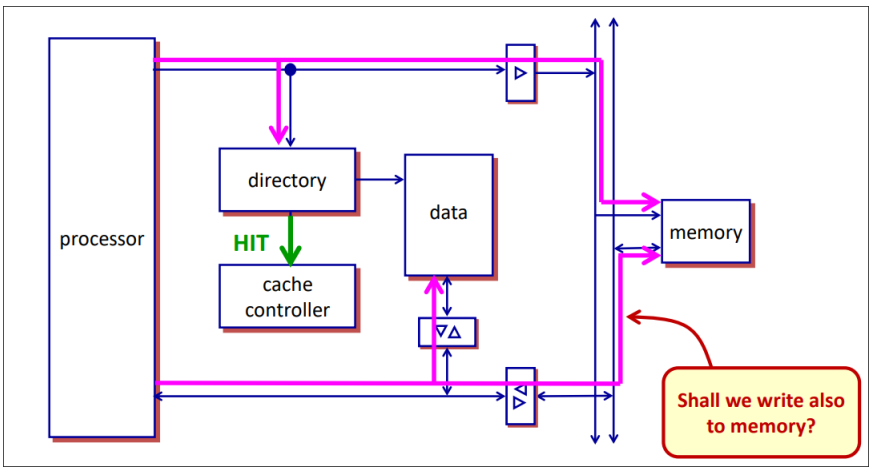
If addressing is **by byte** and **word size is 2^n bytes**, the **n least significant bits** of the address represent the byte offset and are thus **irrelevant**. The last two bits of the address are **internal** to the processor and do not even get to the memory system. As we have seen in Section **??** 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).

4.1.1 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

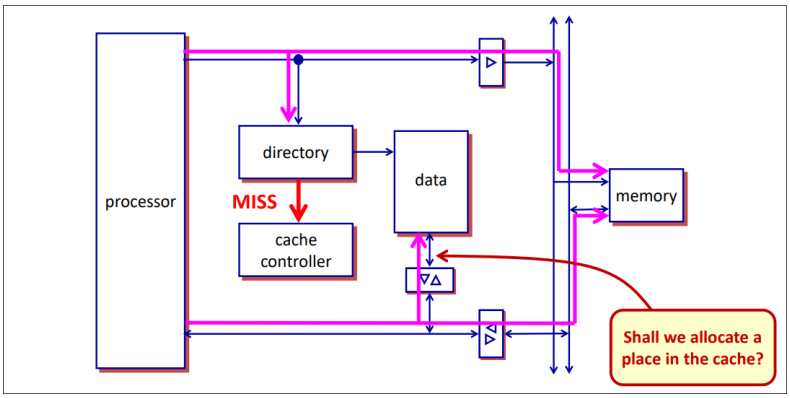
There is a couple of way to fix this issue:

- **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

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**

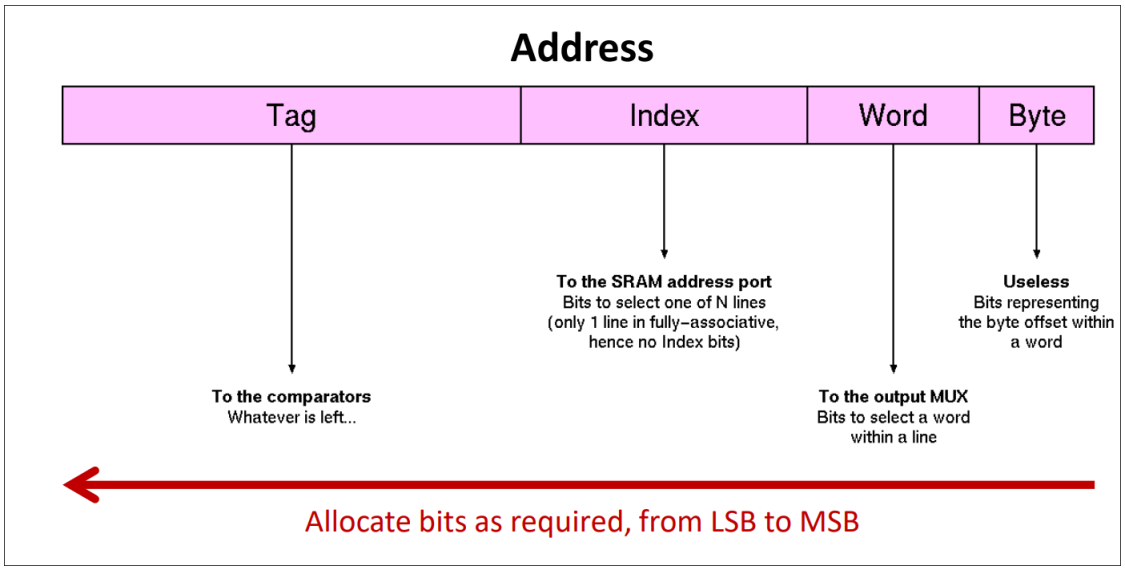
1. **Compulsory** → Missed that would also happen in an infinitely large fully associative cache with the same block (also called **cold-start** misses or **first-reference** misses)
2. **Capacity** → Additional misses that occur because the corresponding block has been evicted due to the **limited capacity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



4.2 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-**

Those are examples

4.3 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

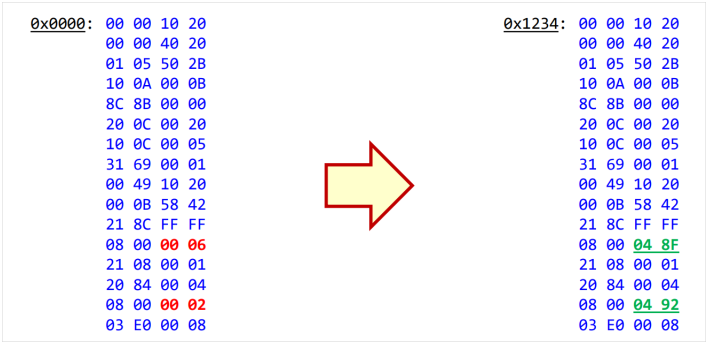
```
0x0000: add v0, zero, zero # v0 = 0
                add t0, zero, zero # t0 = 0
0x0008: sltu t2, t0, a1 # t2 = (t0 < a1)
                beq t2, zero, 0x003C # if (!t2) goto fin
                lw t3, 0(a0) # t3 = mem[a0]
                addi t4, zero, 32 # t4 = 32
0x0014: beq t4, zero, 0x0030 # 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
```

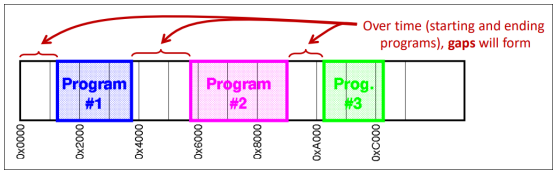
The relocation **must** take place at a **binary** level, not **assembly code**.
We need **relocation tables** to know **where** are the addresses to change



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 :

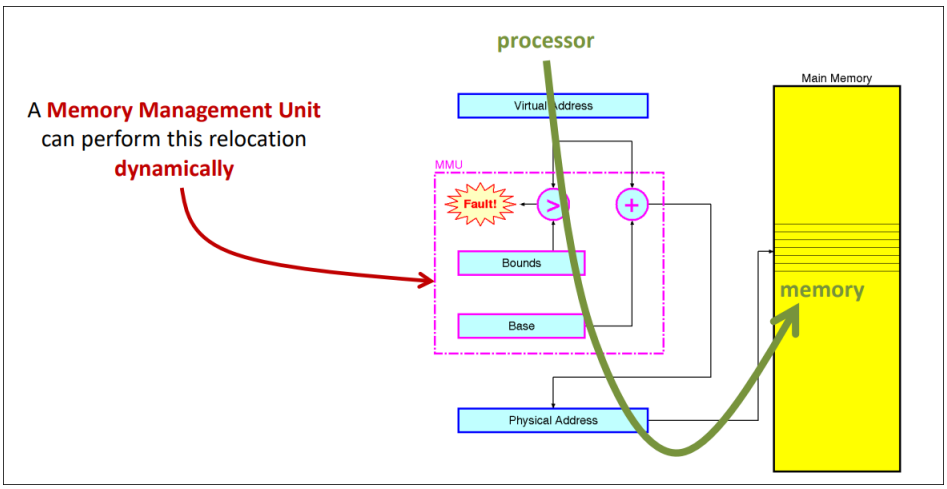
- Large amount of work to do at load time
- **inflexible** → cannot be changed later

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.))

> 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 **??**

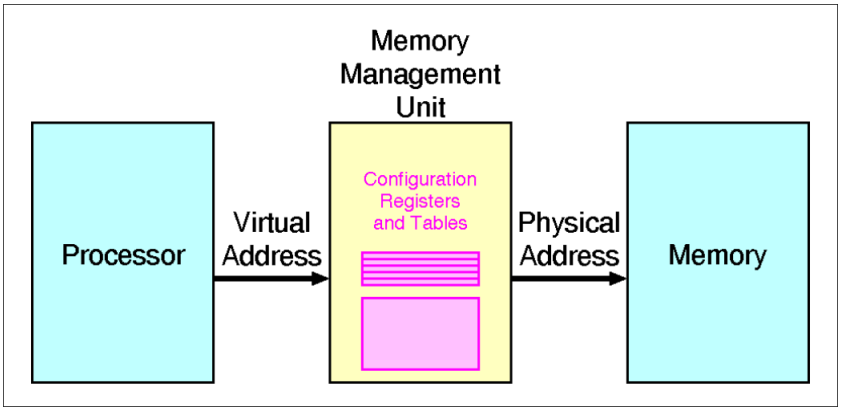
Relocation in hardware:
Base and Bounds MMU

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.

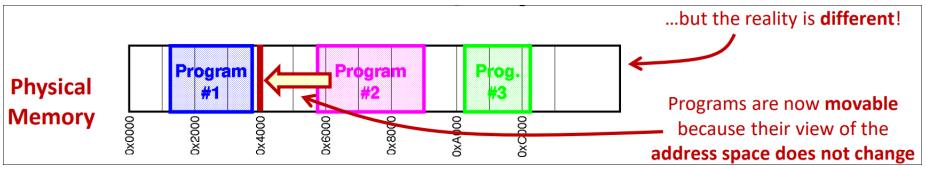


Virtual Memory	This is a memory that the OS allows a program to believe it has.
Virtual address	Conventional address used by a program → the MMU must translate it into physical address at the time of an access
Physical Memory	Memory actually available in the computer
Physical Address	Real location in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

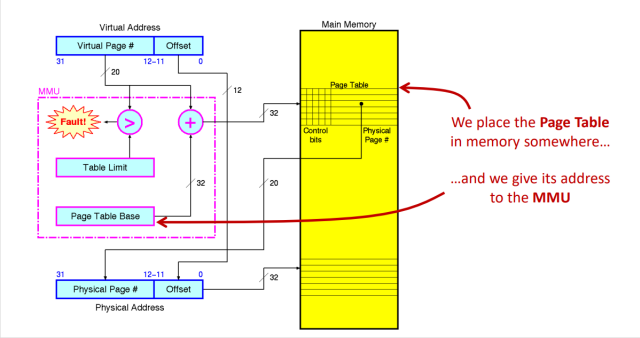
Virtual and Physical Memory

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 **??**. 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.



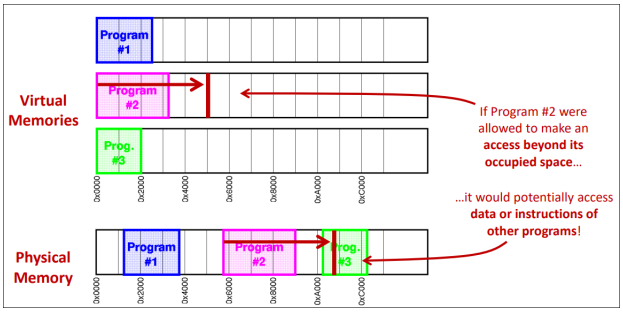
If we take the example of the stack of before, the issue was that when we were accessing returning but the memory was shift then the program was lost. However now, even when we are changing anything in the physical memory, the stack from a software perspective stays the exact same. The MMU handles the 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

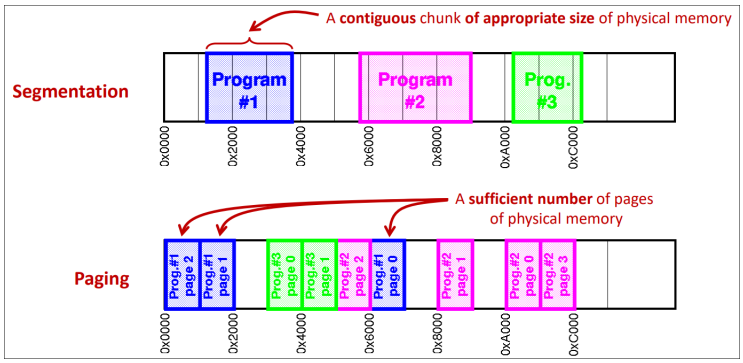
Segmentation

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

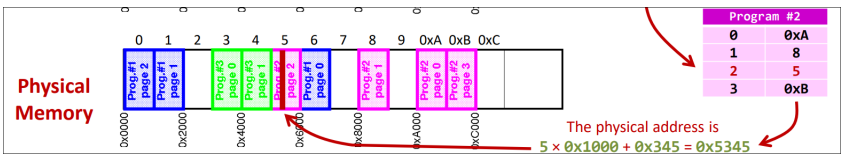
Paging splits the memory in equal small block (e.g., 4-64KiB) and assigns as many as needed to each program.



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

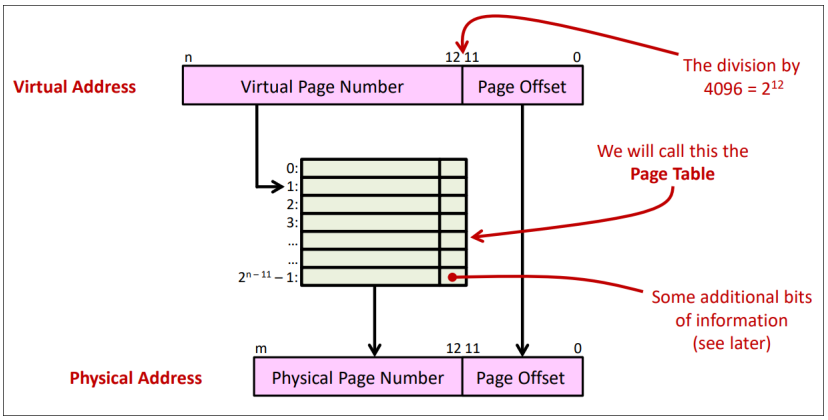
How do we translate?

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?

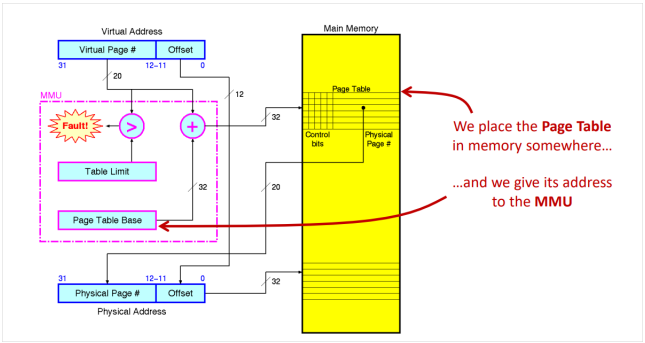
If the length of the page is a power of 2 then this has became totally **trivial**: All we have to do is to take the first n bits for the offset and the $32 - n$ last bits as the input for the page table.



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

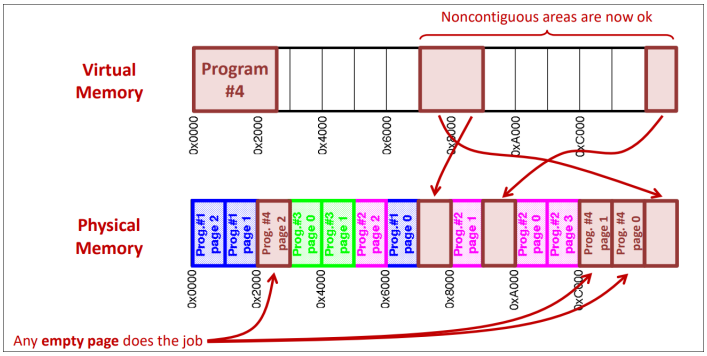
Address Translation in a Paged MMU

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

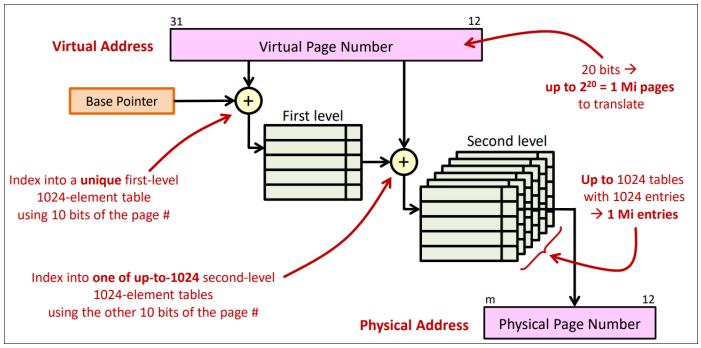
Page table could be very large, for instance 64GiB of memory of memory in 4KiB pages requires 2^{24} entries or approximately **64 MiB**.
For a program that uses only a few MB, **most entries are empty**.
There is a big sparness for the table, those table are gigantic but we most of what we'll use is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

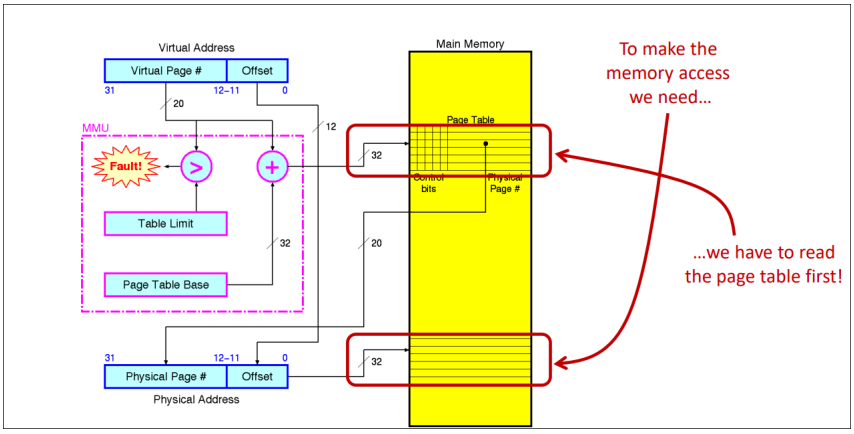


This is kind of weird at first, we are adding a new table, how can we gain time or space by addings something in memory?
But now we don't have to allocate all this space directly in the first place, what we can is only allocate in page tables in memory when it is needed (when the OS decide to allocate).

> 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.

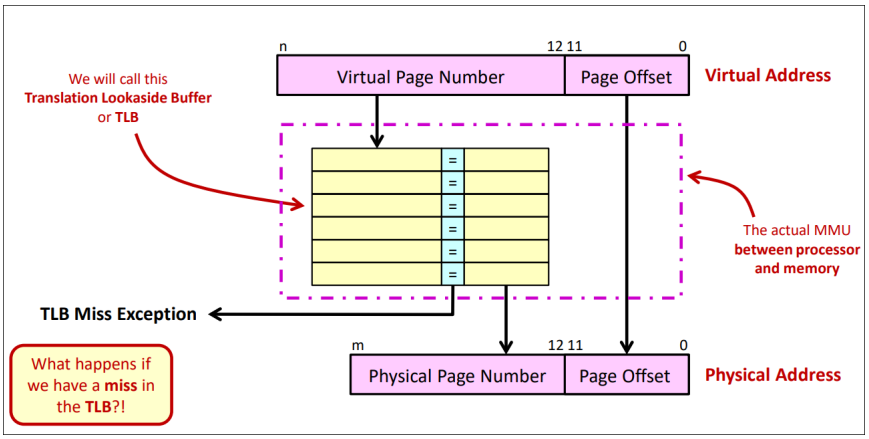
But, Two memory accesses every time?

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):

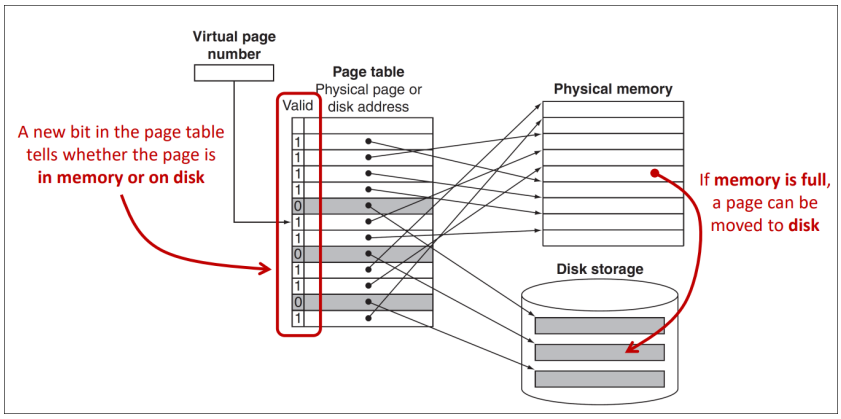
- Valid (to indicate presence in main memory)
- Allocated (to indicae existence)
- Dirty (to indicate a copy-back is needed)
- Used (to help determine which page to replace)
- **Readable**
- **Writable**
- **Executable**
- ...

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

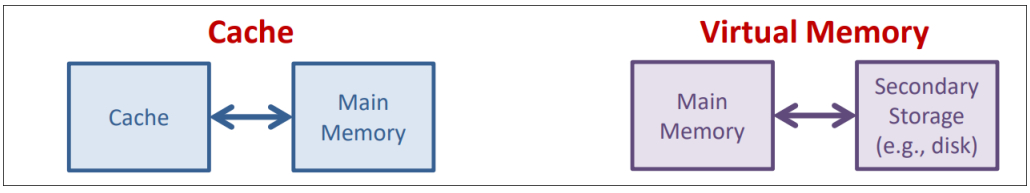
But here we have to rechange our TLB miss because the address can now be also on the disk. this misses that:
When the OS **Reaches the page table** after a TLB miss, now there is a new possiblity: the **addressed page is on disk**

- 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

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

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.

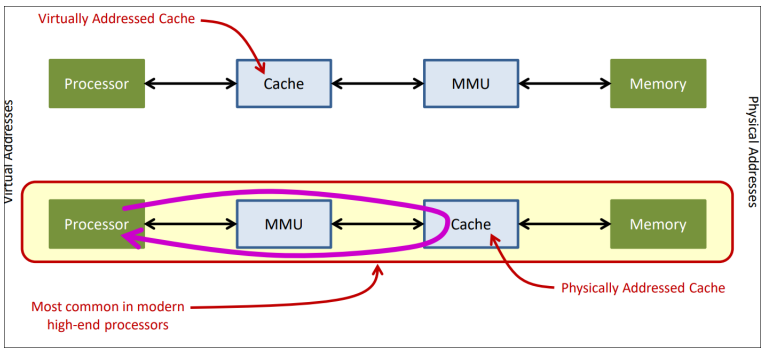
Page table Attributes - Revisited

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
- ...

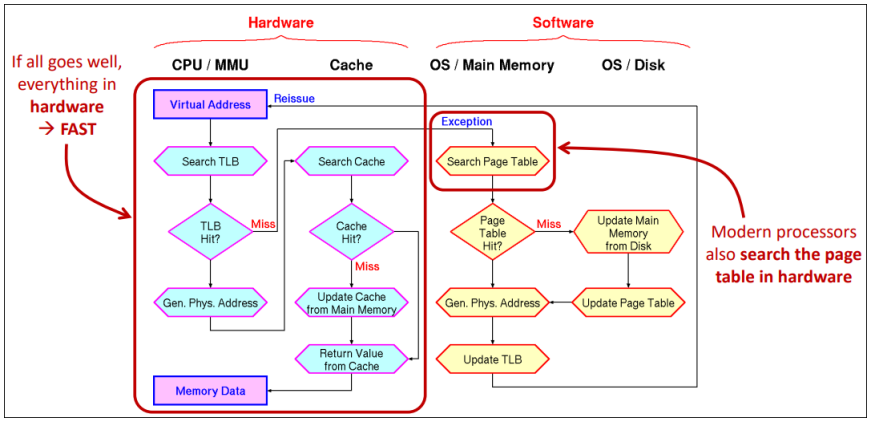
Virtual Memory ⟺ Cache

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

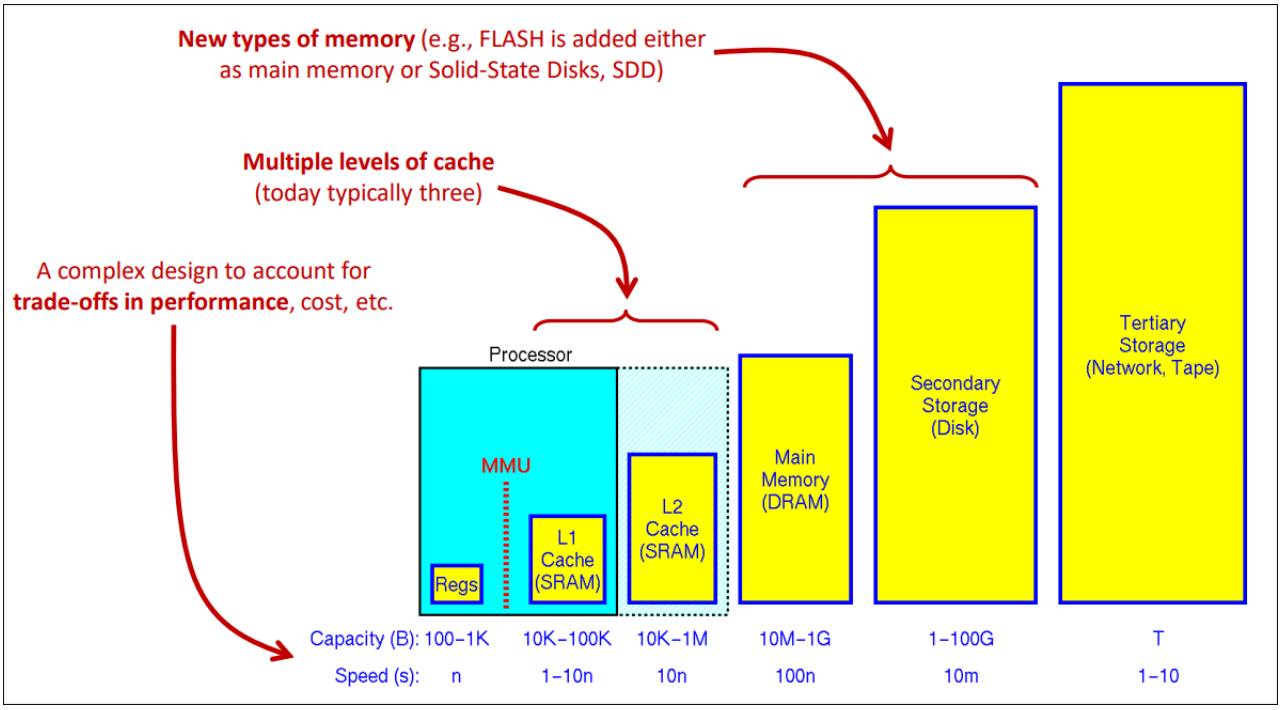


The reason why the most common one is the second is implementation. the MMU translate virtual address into physical address which is then process by the cache. Therefore the cache is the same for every programs while on the other hand if we go the other way around. The cache **is** different for each 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.

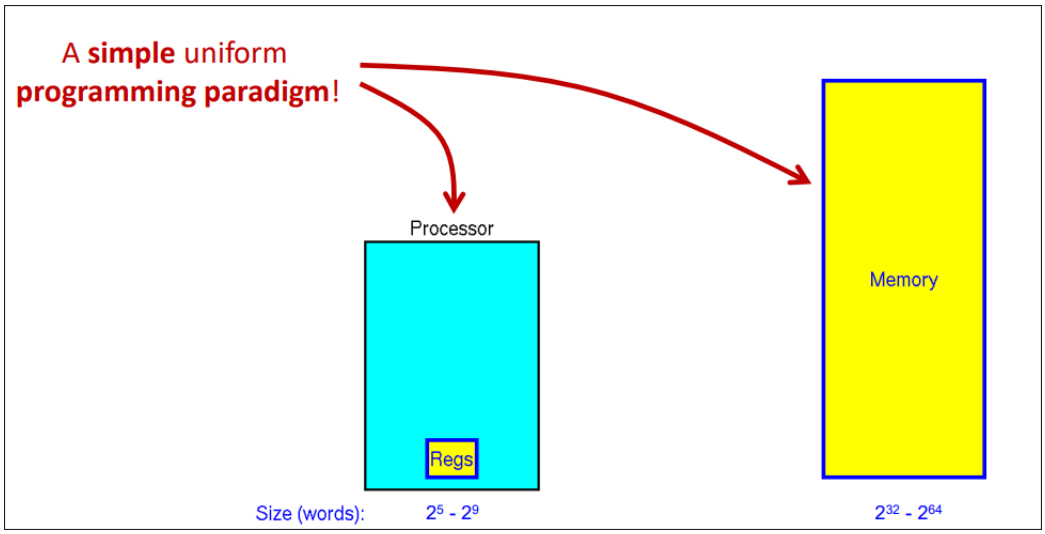
TLB Misses, Caches misses and Page faults



Overall Picture: The System Side



OVerall Piccture: The Programmer Side



4.4 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

4.5 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 ta-**

ble 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:

$$0x3990B0C48$$