tricky3.446Processor tasks on Exceptionssection*.67

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, 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

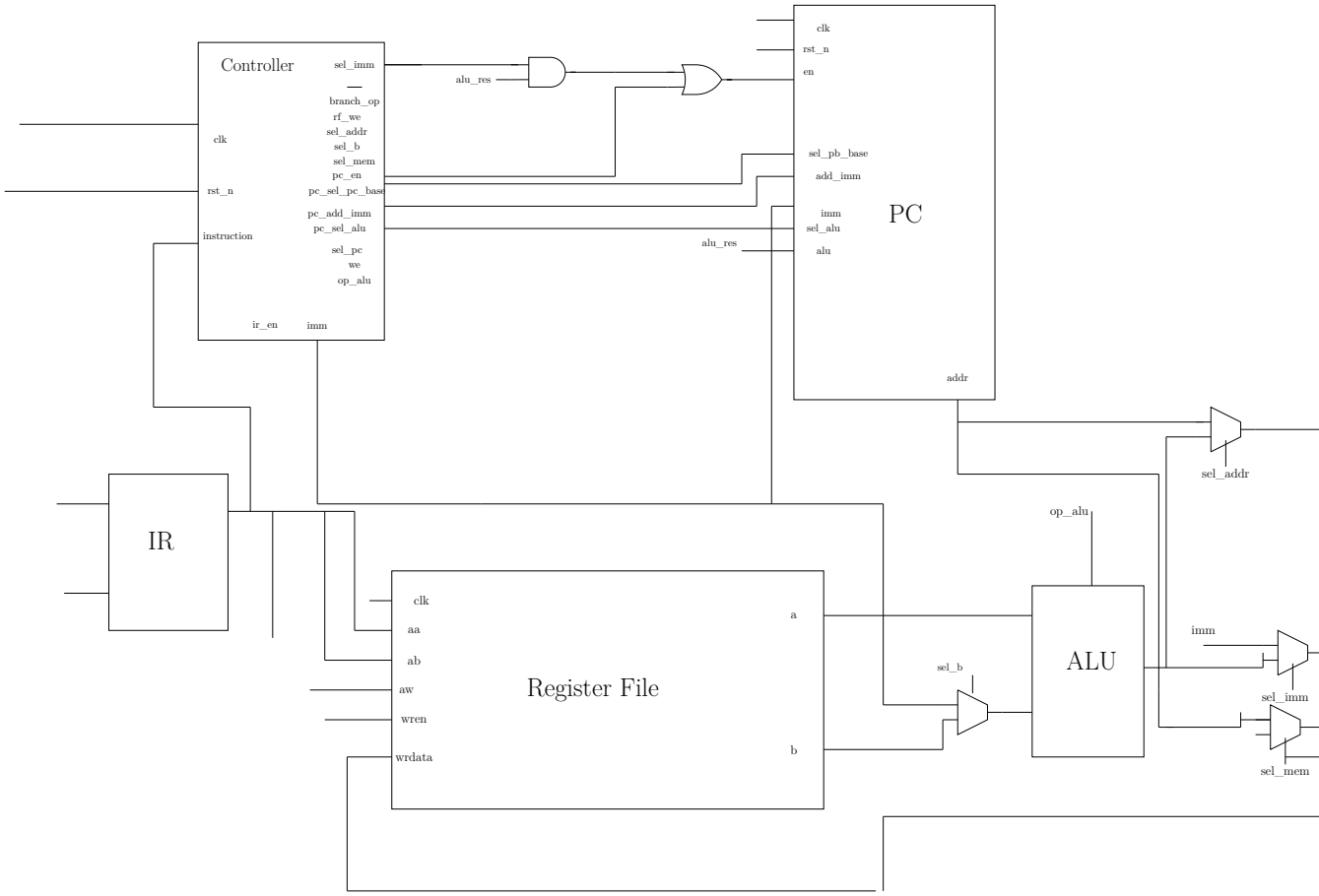
The course is divided into three parts:

- **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

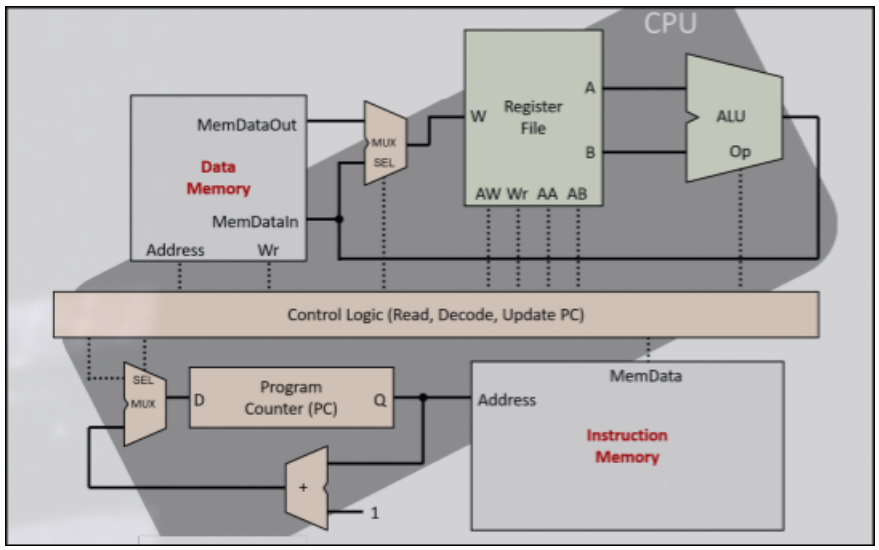
The goal for the beggining of this course is to go from "high-level" perspective to the bottom of the iceberg. First let us look at a piece of code (`c`):

```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);
```

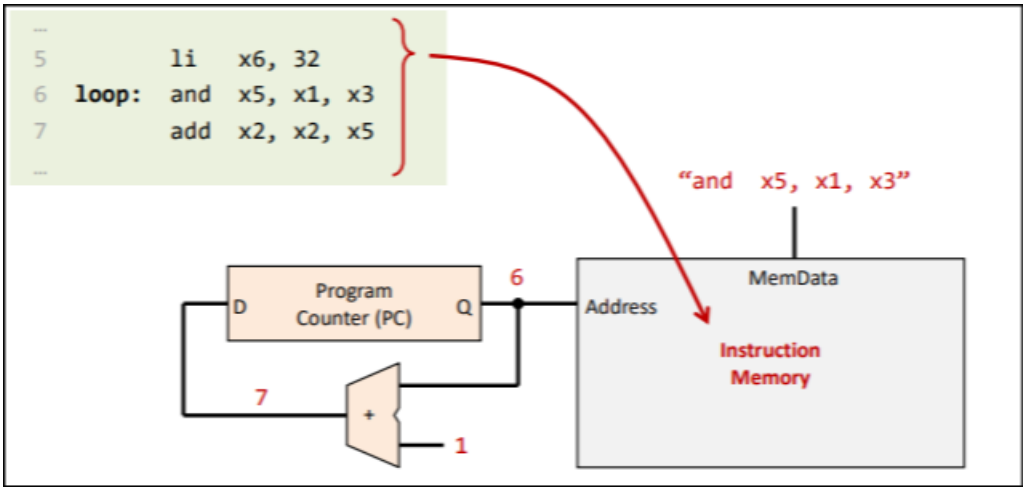
Here we can see that we have variable with expressive names (that we can choose). each variable has a type, the computation we are doing `result + temp` looks like a mathematic formula, the control flow we are using is very intuitive.
In the case of those high-level language, we have an "unlimited" number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

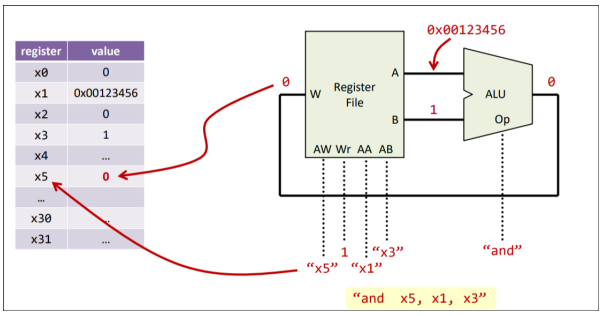
As we can see we have a much more rigid format: we really have a sequence of numbered instructions that is executed line by line. For each instructions, we have an *opcode* that defines the effect of the instruction. Each *variable* has a fixed name and we only have one form of control flow. The question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

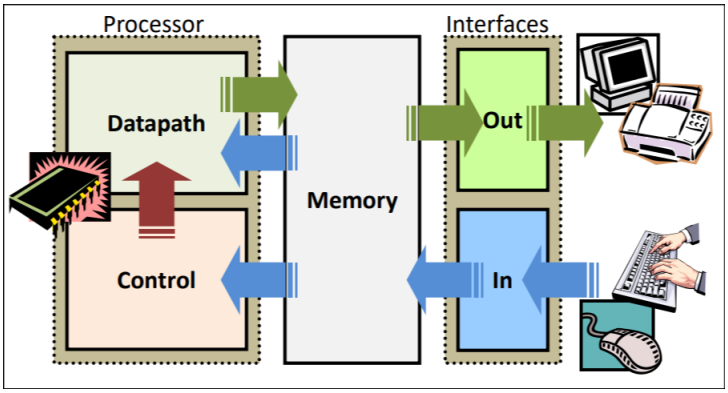


After that the instruction has been fetch, it is processed in the Control logic and then read/write etc... into the register file, and give the information (the opcode) to the ALU for it to know which operation to perform.

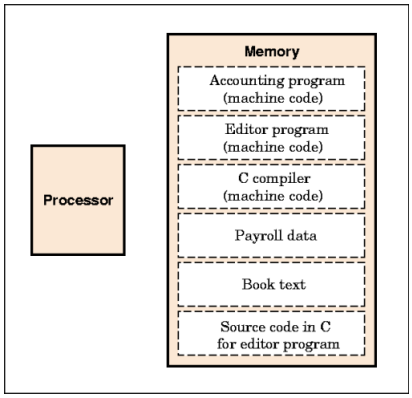


The five classic components of a computer

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).



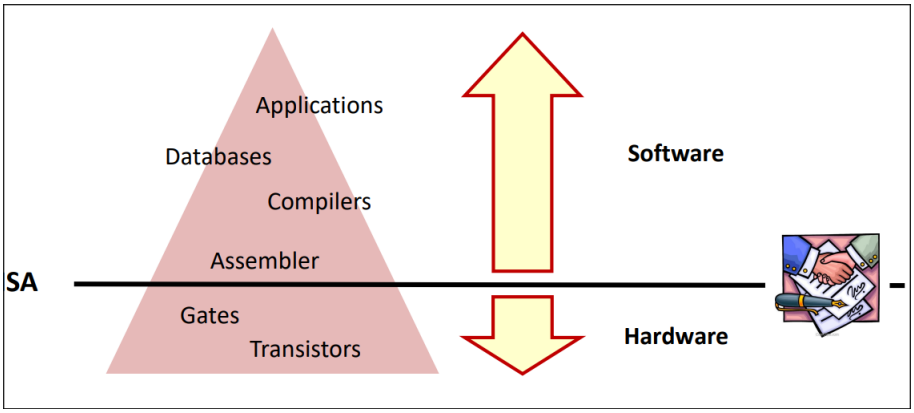
Now a good question to have is: how to decode and encode those instructions and in the mean time, also what makes a good encoding?

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 Functios

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:

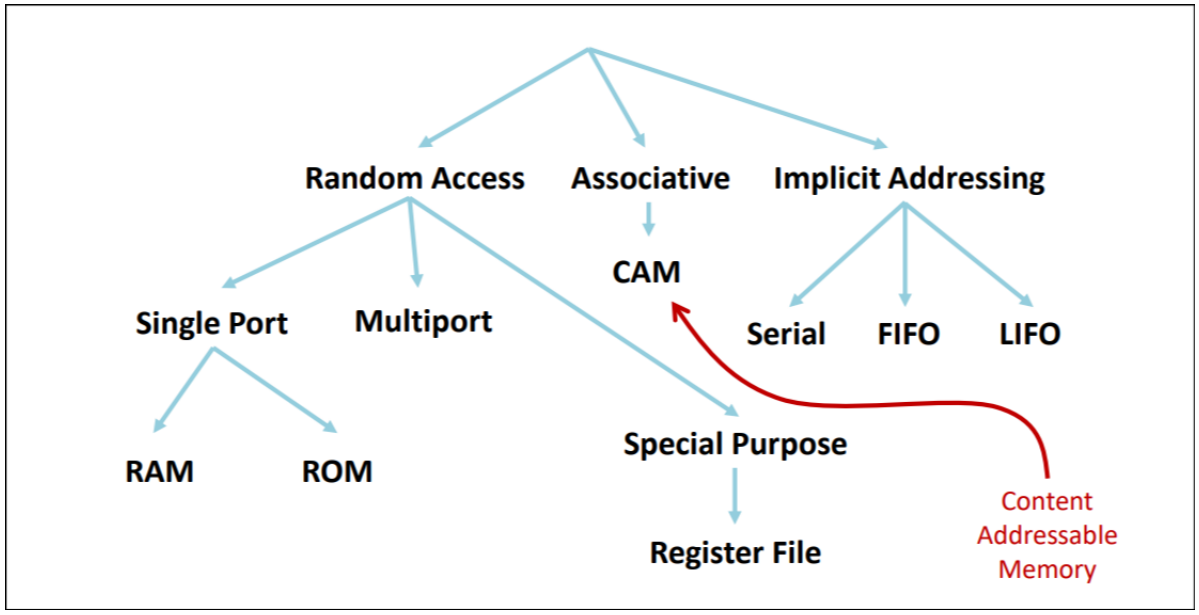
- Memory can be **very slow** $\implies$ Caches
- Memory is *finite* (relatively small) $\implies$ Virtual memory
- Memory can make an **ISA too complex** $\implies$ pipelining

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

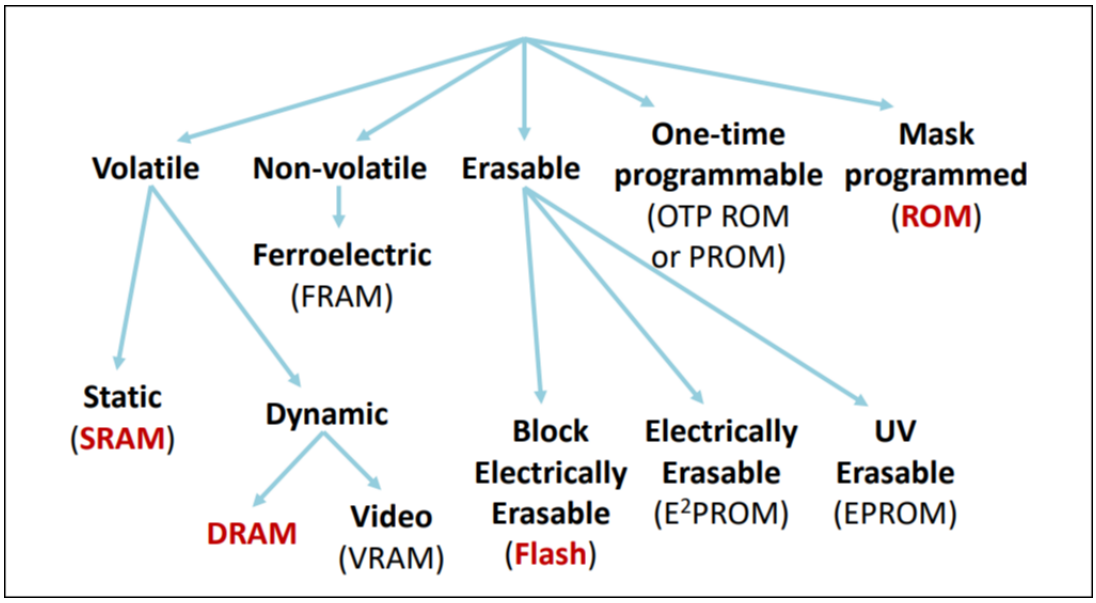
- SRAM, DRAM, EPROM, Flash, etc.

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

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

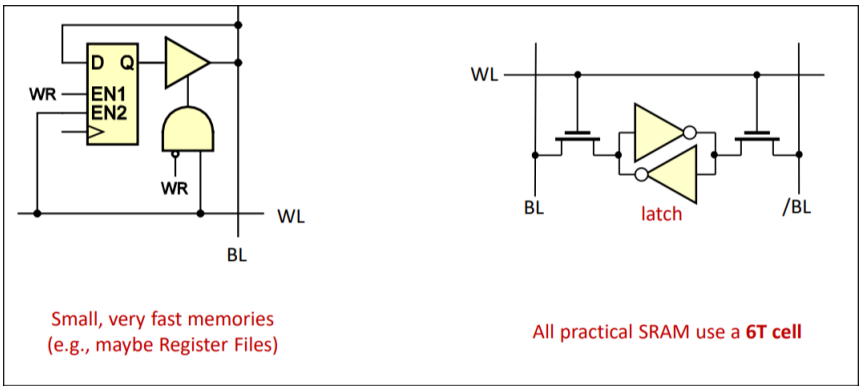


We have here all the type of memory that can be used, What we use when we program are Random access memory, this means that the memory can be accessed with an address. In the tree of random access memory we have:



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

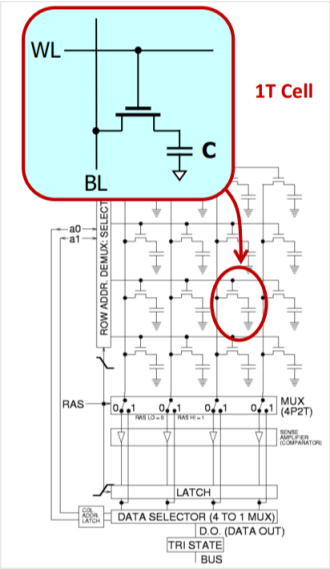
it is stored for infinity.
The only way to write is to "*shout louder than the current signal*". Imagine we currently have a 1 as the output of the latch and I want to put a 0. If I shout 0 louder than the 1 while connected, it will have a short circuit... and this is bad. **However** what is going on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

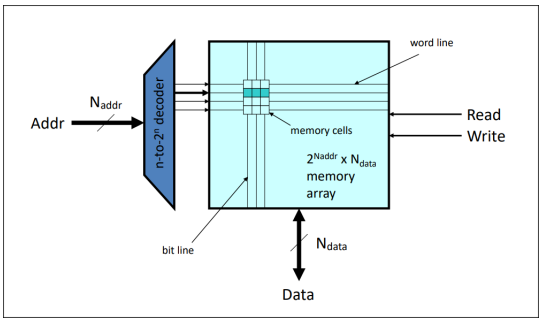
DRAM

- 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.

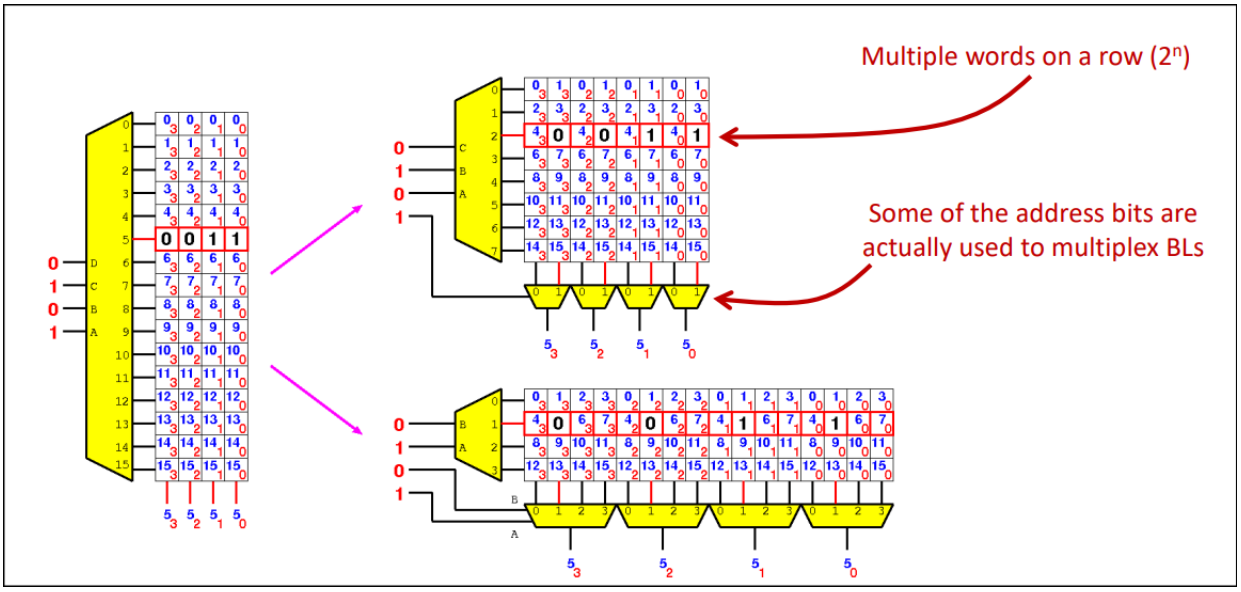
So imagine, if we don't go in each cell every 60ms then we lose the information, but we have other things to do? So how do we do it? - we have someone else refresh them for us. The memory controller is reponsible for refreshing the contents of the DRAM instead of the CPU.



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:



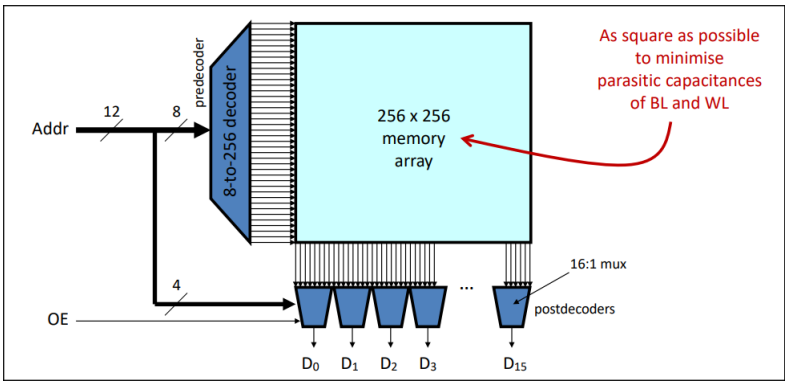
Here we have 3 ways to do so:

- 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

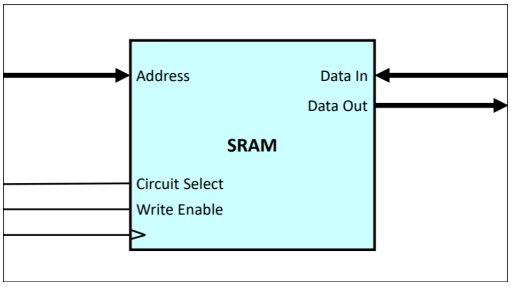
$$A = rc$$
$$\frac{A}{c} = r$$

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.

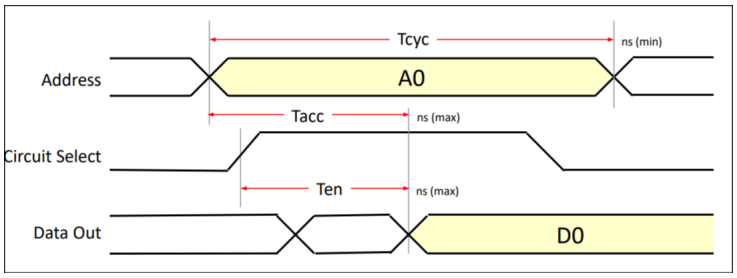
Static RAM typical interface This is the typical synchronous SRAM that we have already seen before:



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

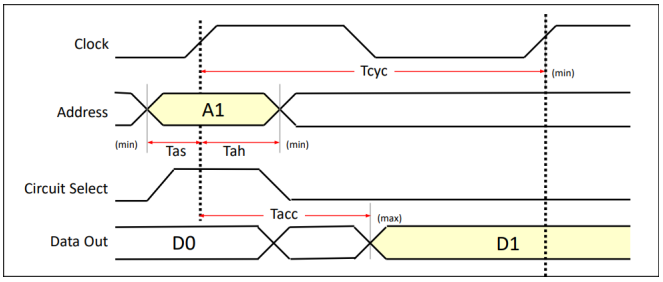
- Enable the memory → assert the address → wait for the data
  - Data out is available after a combinational delay T_{acc} = Access Time
- Maximum frequency is limited by the minimum T_{cyc} (time for a cycle, time for us to be able to change the address)



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

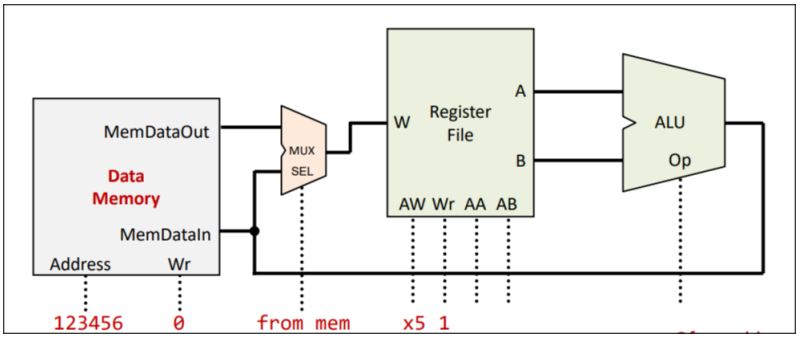


Load and store instrucitons

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:

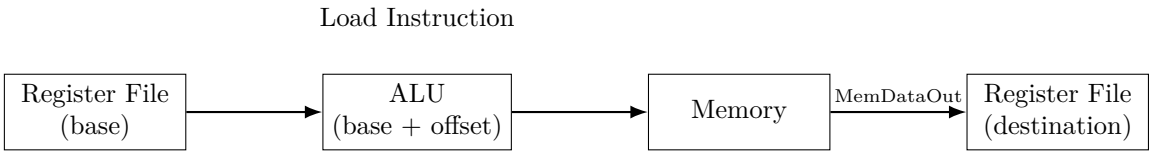
```
lw x5, (123456)
```

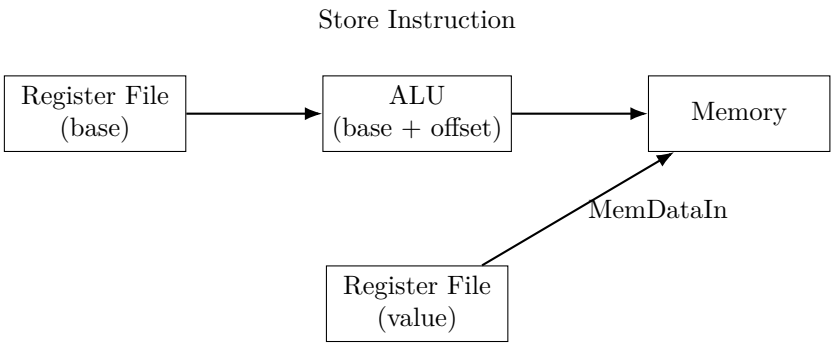
This is not a RISC-V instruction but bear with us. (I am not sure that's a saying...)



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]
```

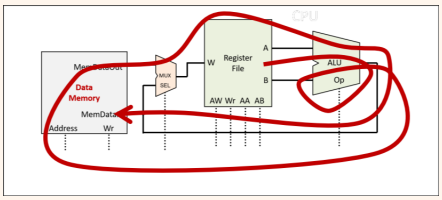
However those instruciton **does not exist** in RISC-V.
RISC-V is designed to have two world: one for accessing memory, and one for the logic/arithmetic etc. We cannot mix them together.
However in x86/x64 we can do this:

```
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:

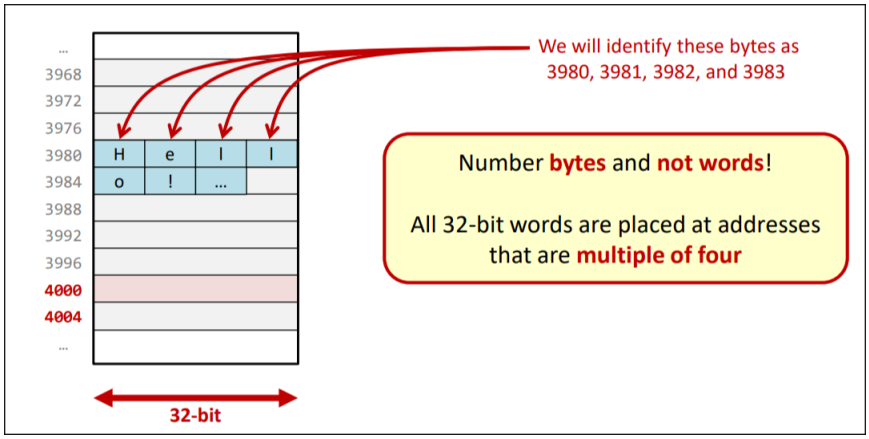


We would need to go like four times to the ALU, the fact is that **this is possible** however it makes it hard to optimize the processor.

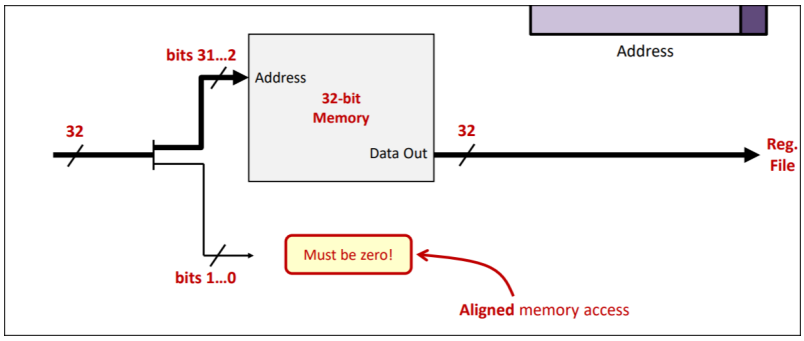
The question behind all this is how does intel can still be as famous as they are now with instruction that looks like this and that cannot really be optimized? There is still the majority of the processor to this day that are intel processor (even if amd is better...), we will see this in a couples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

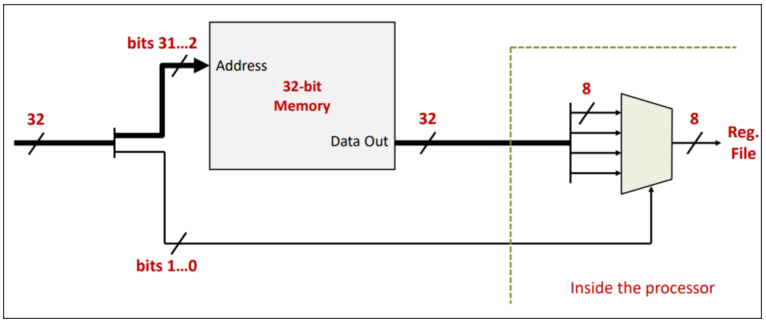


Loading a word (`lw`) and Instruction For instance, if we are interested in word, we cannot look for the word at address 3981, this would'nt be a word.



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 byes (`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

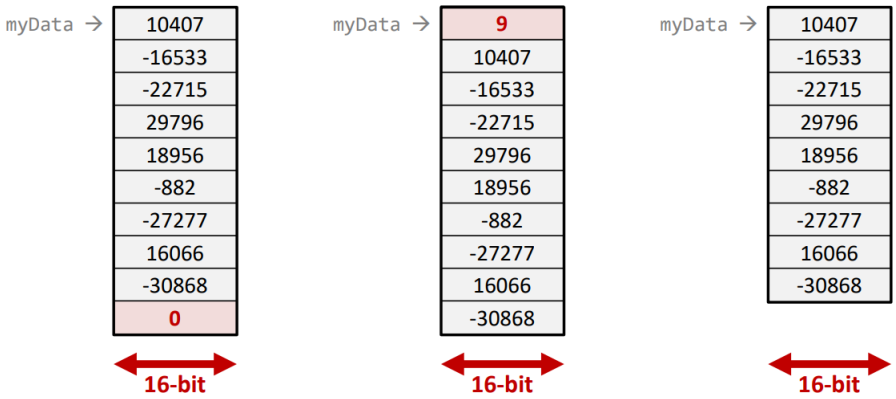
Data structures is one of the main concept in computer science, even in this course we have already talk about it (the stack).

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).

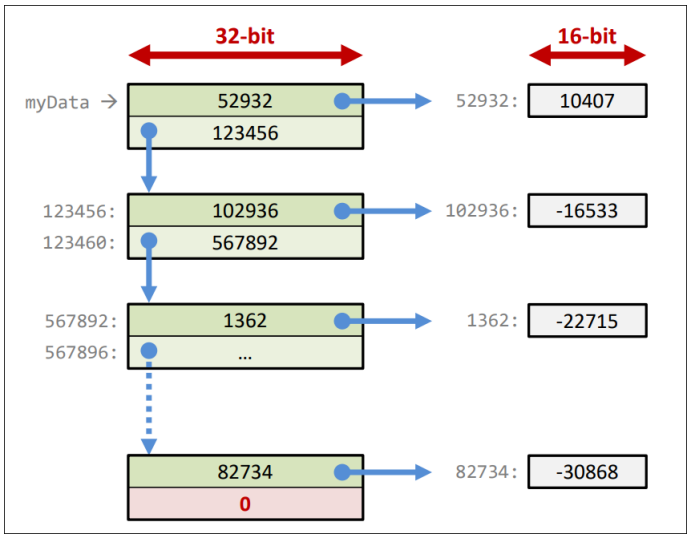
```
addi_positive:
        li t0, 0 #t0 will hold the sum (initialized to 0)
        mv t1, 0 # t1 will hold the array index

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.

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)

$$< a_{n-1}a_{n-2} \cdots a_2 a_1 a_0 >$$

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

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

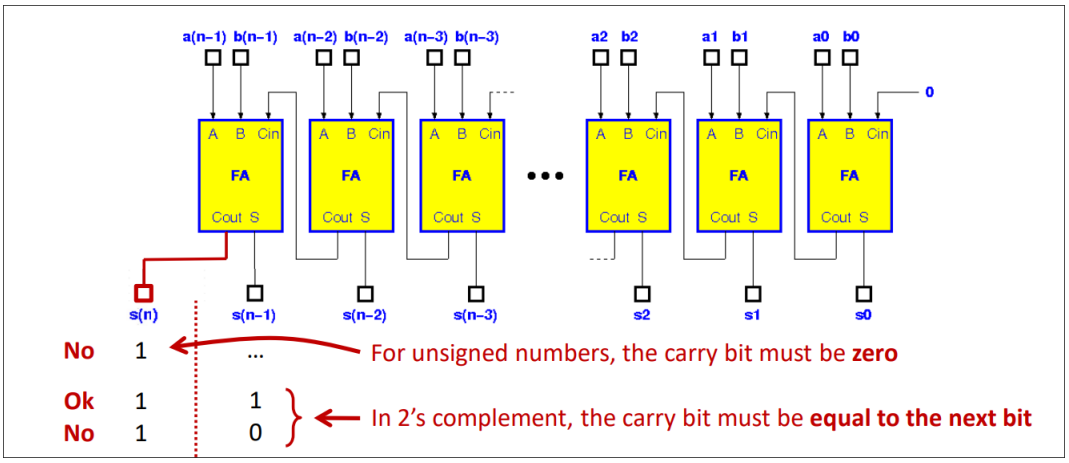
For the next part of this course I am gonna skips this because it is a big review of what we have already seen in fds, I am just going to write what I find intresting **for me** which is not necessarly the most important thing.

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

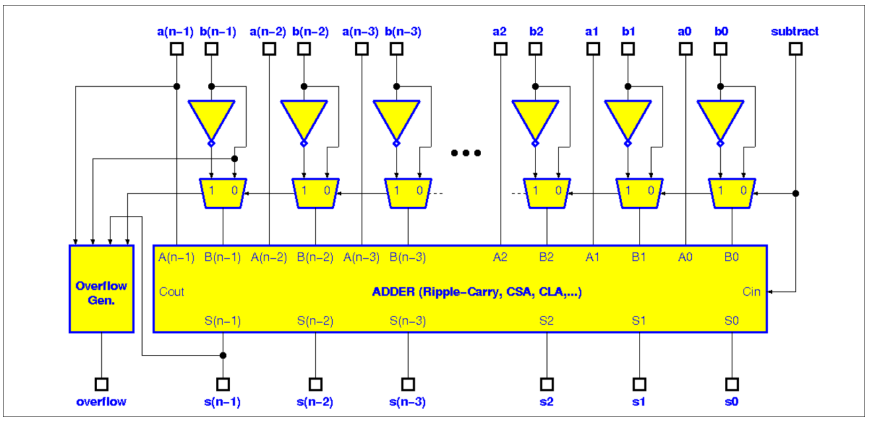
$$-A = \overline{A} + 1$$

To prove it:

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

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 $\neg b_i$



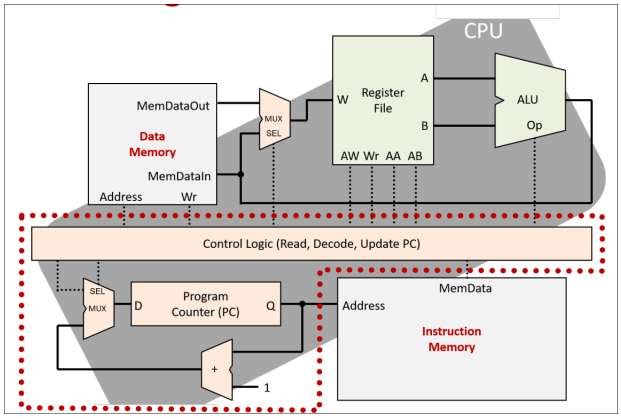
as we can see this allows us to put the substraction and addition into the same module.

Chapter 3

Processors, I/Os, and Exceptions

3.1 2a. Multicycle Processor

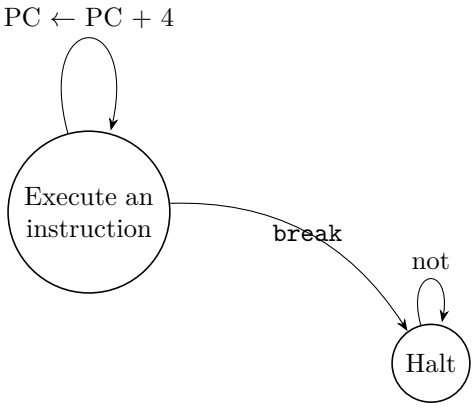
In this section, we will more go into the detail of the **hardware** behind the cpu. Especialy multicycle processor.
As seen before, the CPU has more than one part:



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:

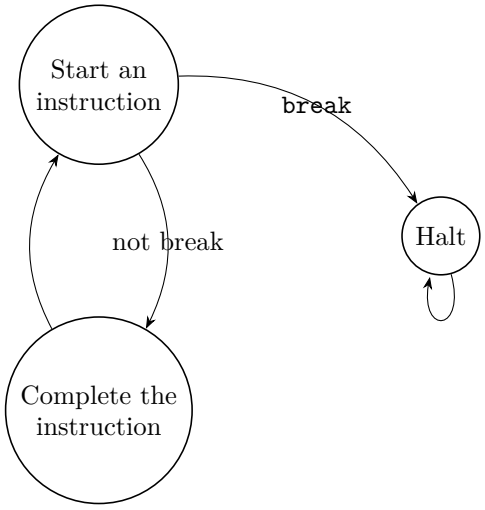


Remark If someone knows how to makes the break here above or below the line I will be curious how.

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 frequncy: **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:

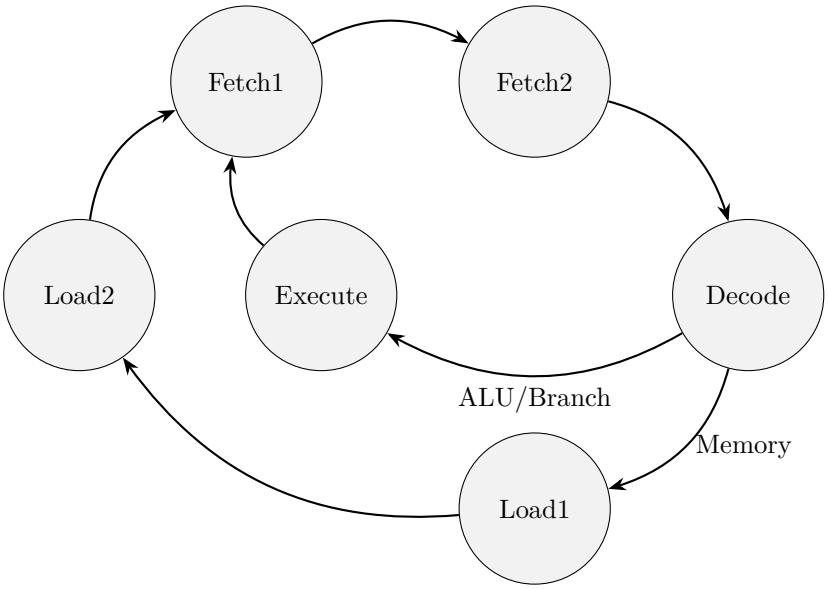


The question we must ask now is: Did we gain anything?
At the moment not really, before we had 1 instruction **per cycle** at frequncy F. Now, we have 1 instruction every **two cycles** at frequncy $2F$.

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.

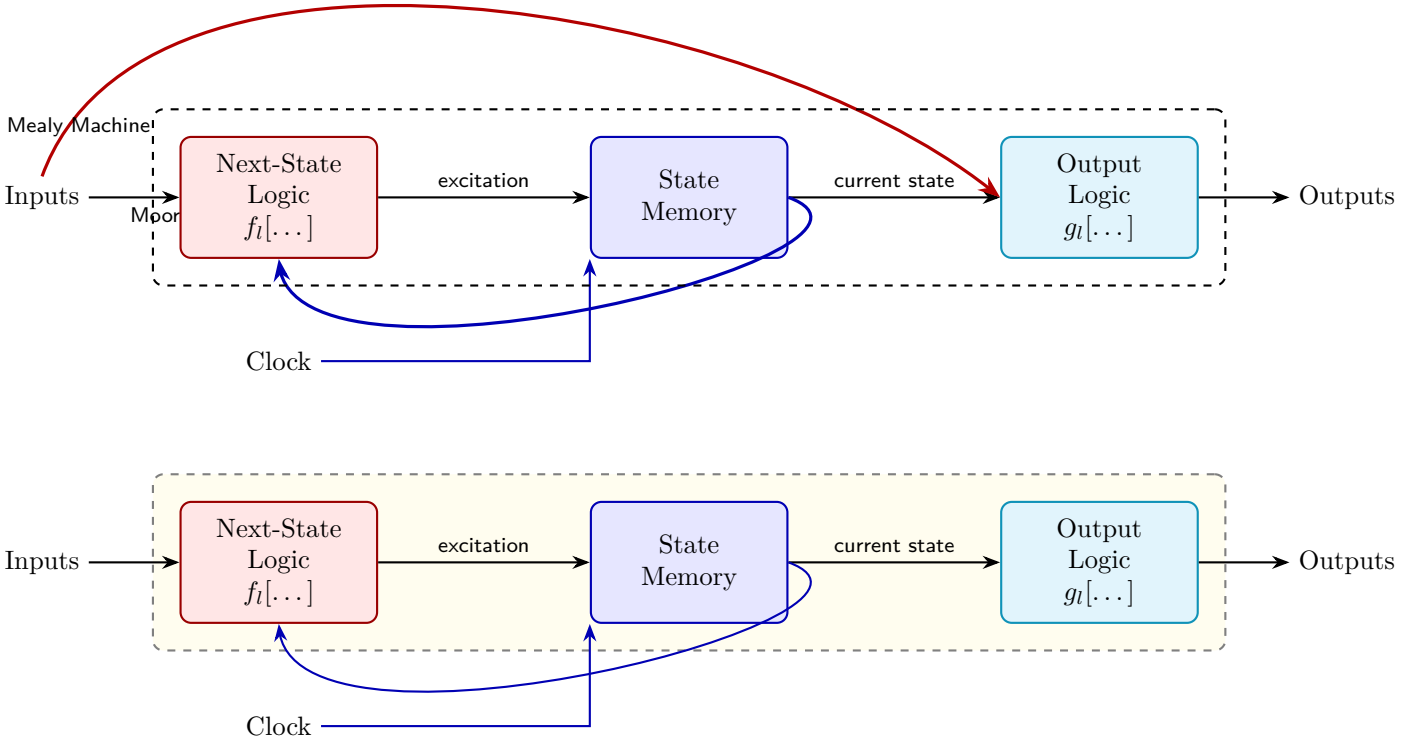
Multi cycle Processor



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 depeneds 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 **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 progressiely 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:

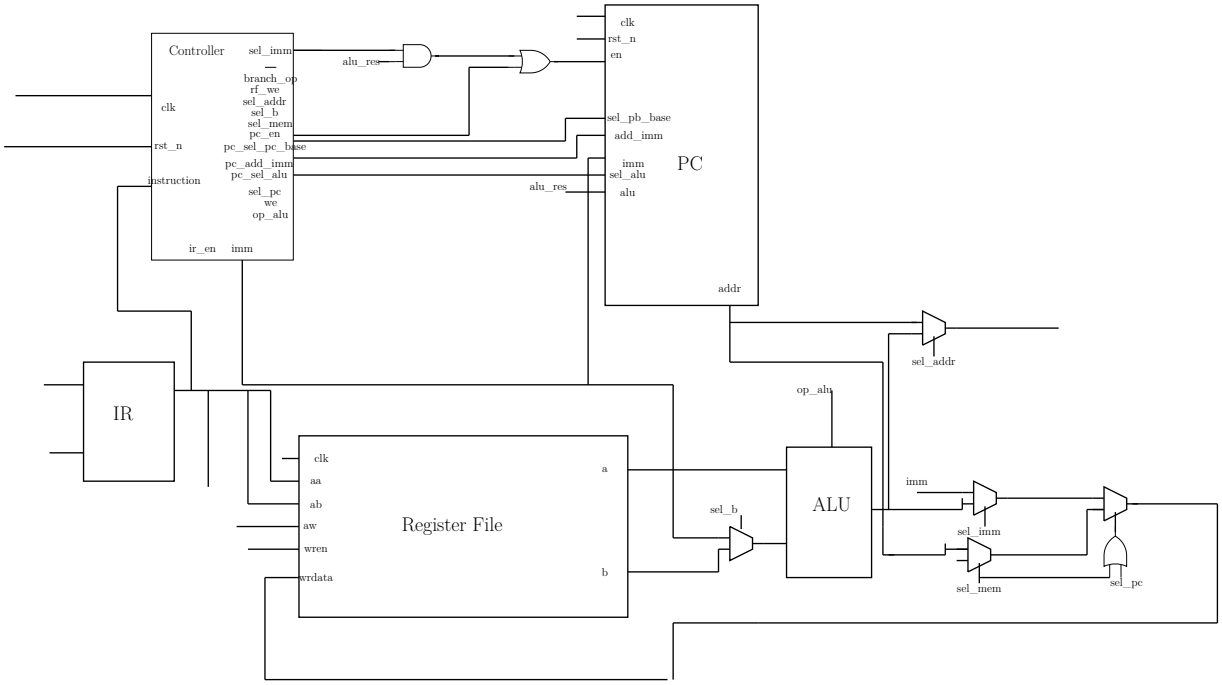
1. `branch_op` $\rightarrow$ informs us of if we are in a branch operation
2. `alu_res` which is the lsb of the `ALU` as said before

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

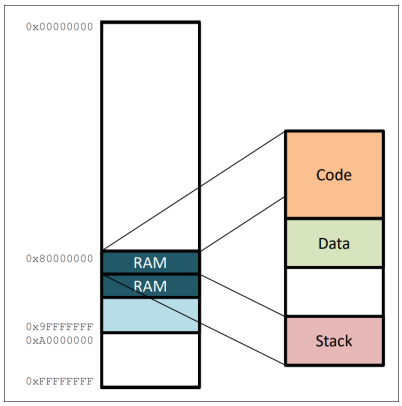
`sel_pc` signal.



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).



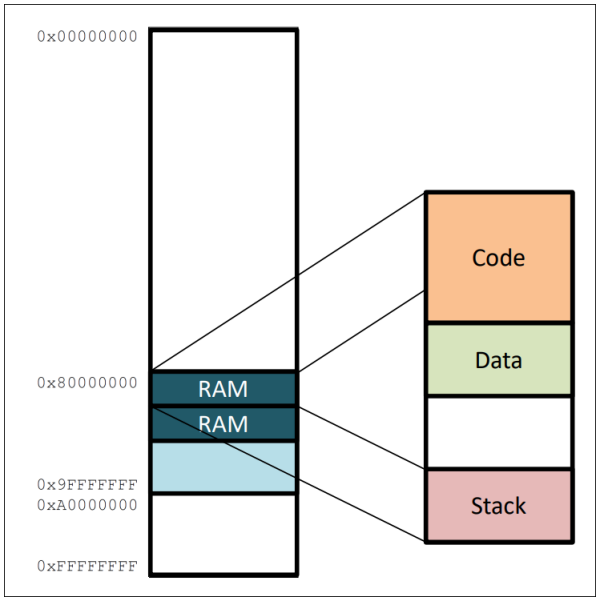
3.2 2b. Processor, Inputs and Outputs

The cpu

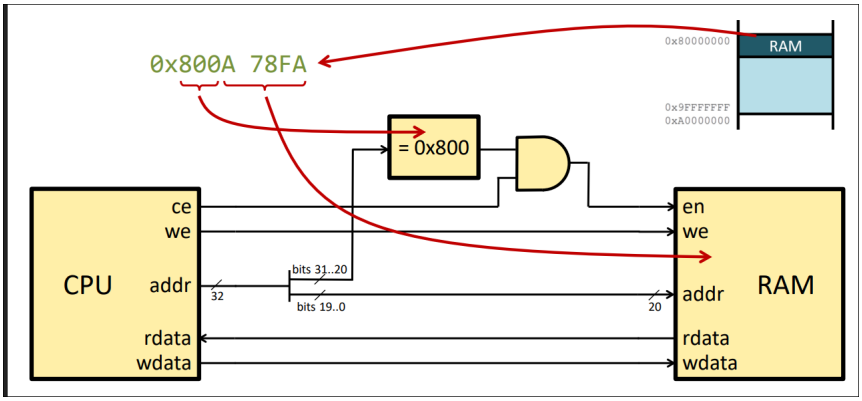
As said before, the cpu is a very **sequential** component (but from now on we may omit the **clock** in diagrams). Therefore here we only have one data bus for read and write in our processor. The question we have is how can we handle input and ouput with only one data bus?

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:

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 the to CPU the port `ctrl-IO` and `port` Which serves as circuit enable and output enable. (this way, we'll know when accessing memory or

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)

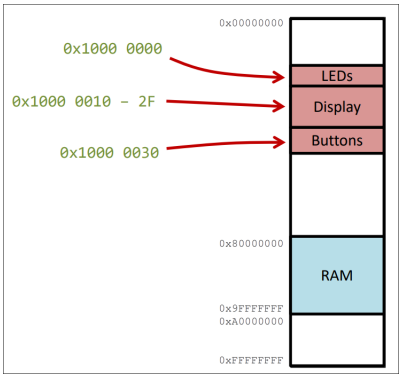
Table 3.1: Approximate data rates for common peripheral interfaces.

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)

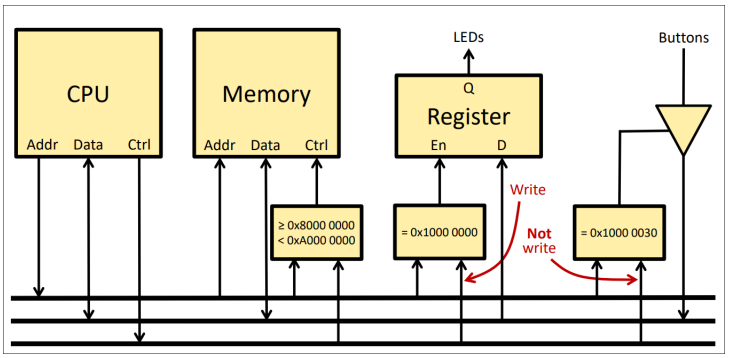
In this way, we don't make any difference between memory and I/Os.



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
```

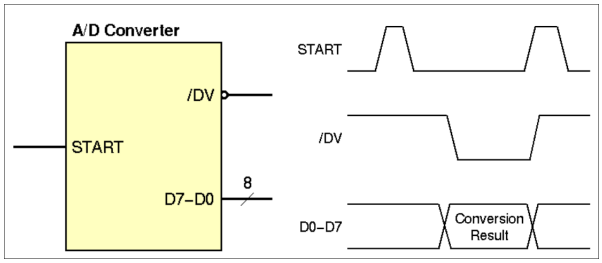
This means that we have a big data bus and depending on the value **the bus** choose between the Memory and the I/Os:



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

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

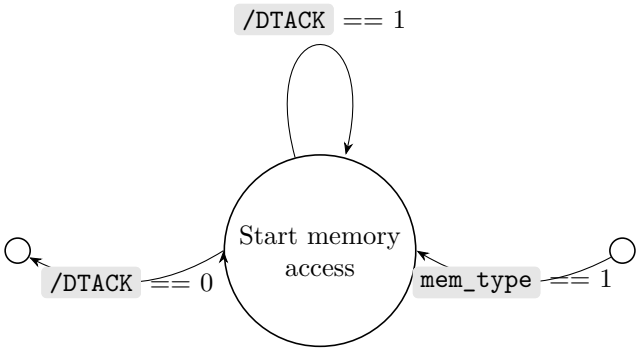
This is similar but not identical to the MC6800

> 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.

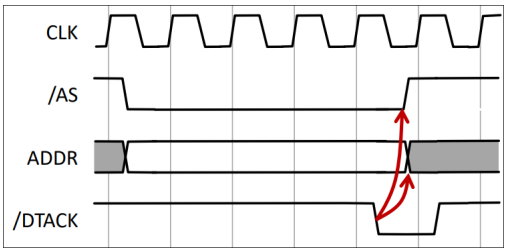
The goal now for us is to create a circuit that is able to connect the A/D converter to the processor (the MC68000). For this we will use a **memory mapped interface**.
We want:

- **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 construc 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**.



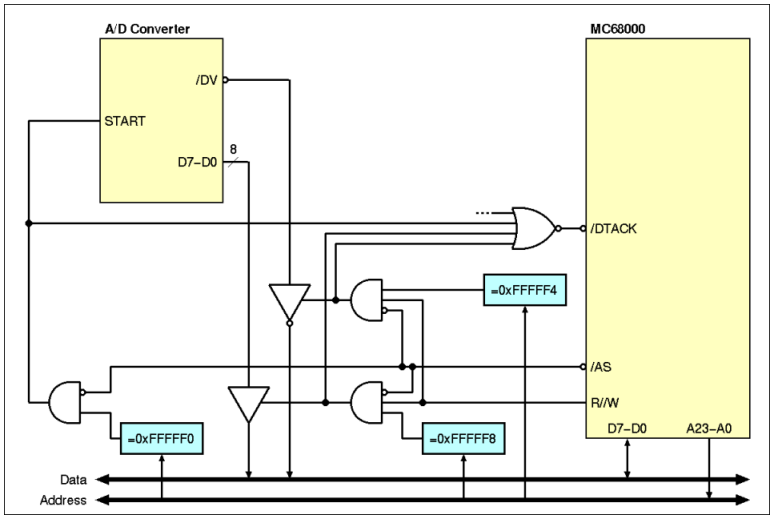


Figure 3.1: A/D converter circuit

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.

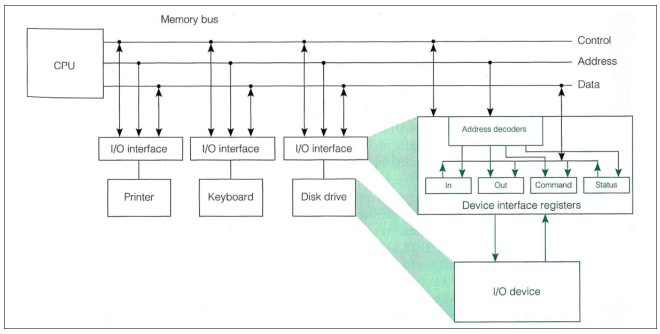
A/D converter: software

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 = 0xfffff0
        sw zero, 0(t1) # start conversion
poll:
        lw t0, 4(t1) # t0 = DV signal
        beqz t0, poll  # wait until done

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

Programmed I/Os

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



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:

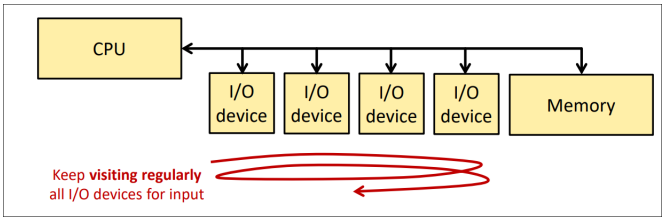
- A **control register** for the processor to configure the UART
  - Bit 7 must be set to 1 for the UART to be enabled
  - Bits 2..0 configure the communication speed (e.g., 0b001 for 9600 baud)
- a **status register** for the processor to check the status of the UART
  - Bit 1 is 1 if there are data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

```
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 responds, we can do our work and **when a I/O** *shouts* we do what he want us to do. We have them **ask** for **attention**

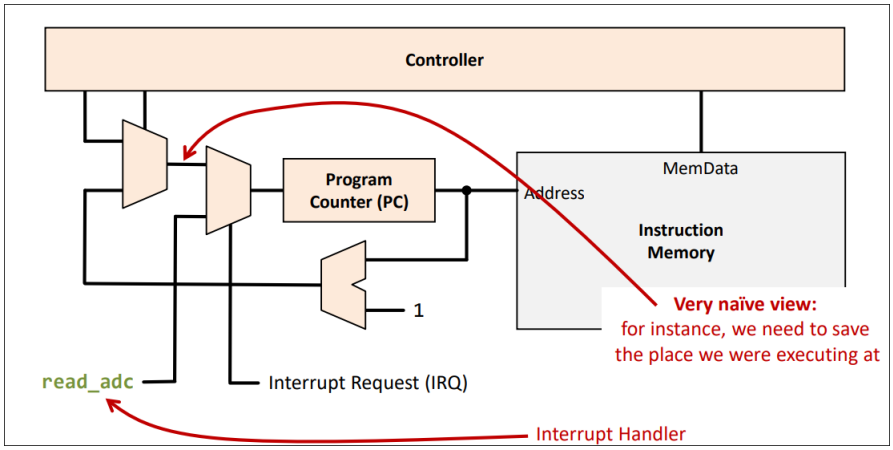
Seen this already in some languages

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

- Callbacks, Action, or **Event Listeners** (PPO), 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

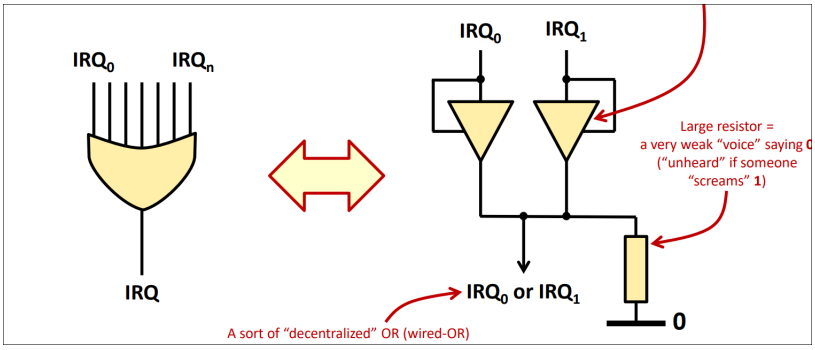


We use the `read_adc` address which is where we handle the I/O interrups. The interrupt request serve to trigger the interruption.
However the is several issues to take care of and behaviours to define:

- 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? Five? Twenty? What ahhpens of the program that was executing?

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 a 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 reverts to the interrupt task

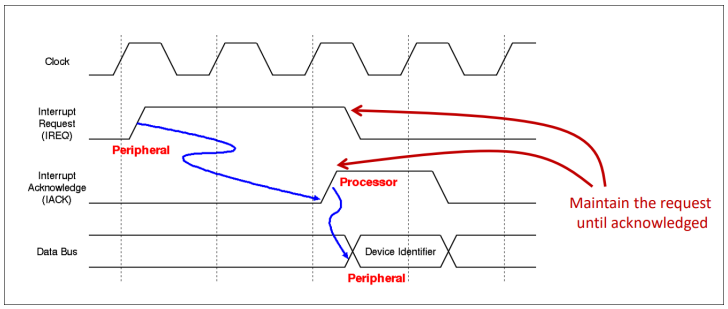


Figure 3.2: tiaming diagram

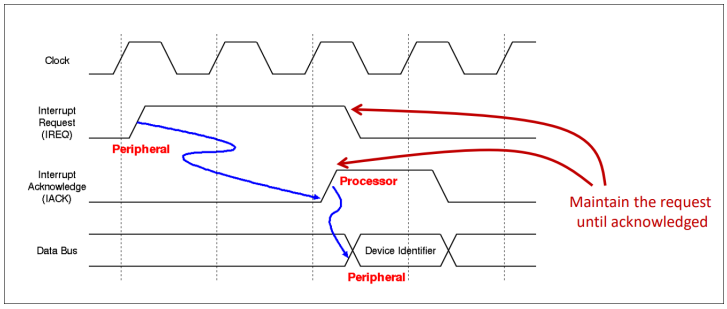
On the software side, it works with the same code we used before for polling. However, the `read_adc` function is now called by the interrupt handler. The interrupt handler stores the address of the program that manages all interrupt requests, and from there, it launches the program that handles the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

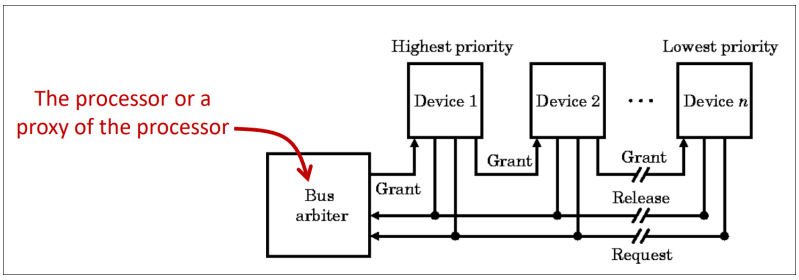
- Anyone places request (IREQ, Request)
- Acknowledge line (IACK, Grant) passed from one device to the next
- Device which wants access, intercepts the signal and hides it from successice devices
- Simple but (1) slow and (2) hard priorities (meaning hard in like hardcode something).



Interrup controller

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

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



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

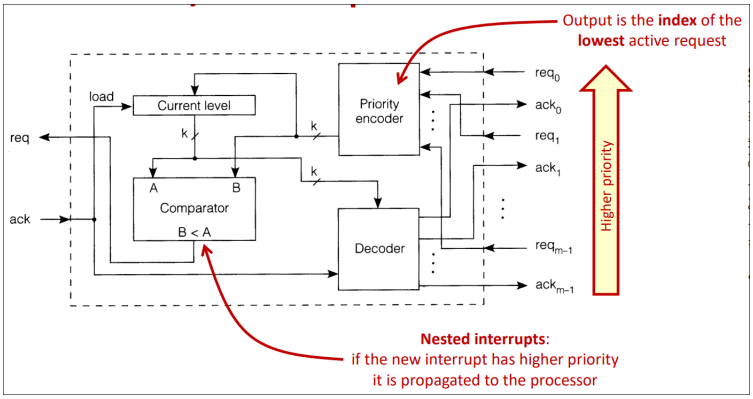
This processor can be done for like 1,000,00 times.

With the DMA

The idea now is to:

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.

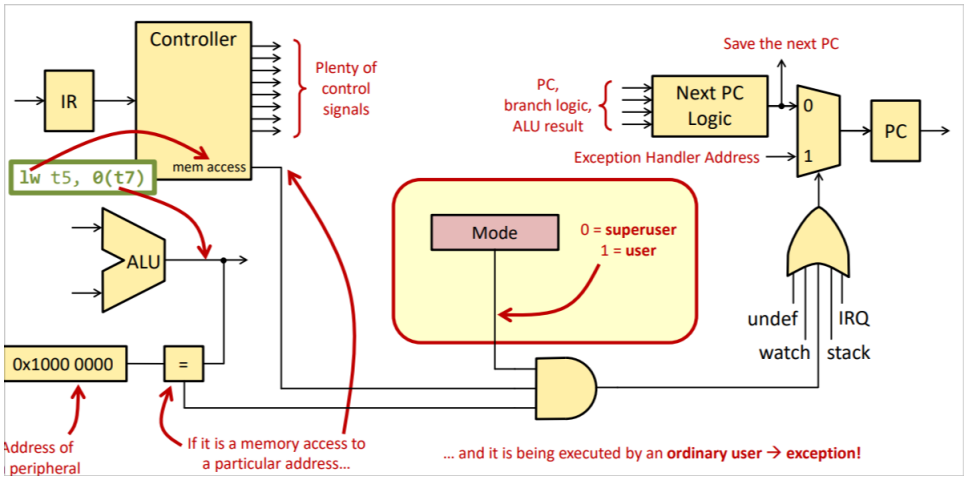
2. The processor acknowledges the timer interrupt and instructs the DMA controller to start a transfer from a peripheral to memory.
3. The DMA controller becomes the bus master and performs the required memory accesses, reading from the peripheral and storing into memory.
4. Once the transfer is complete, 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.

lectureDate2025-10-11 lectureDate — **Cours 10 : Processor and converter**

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**!

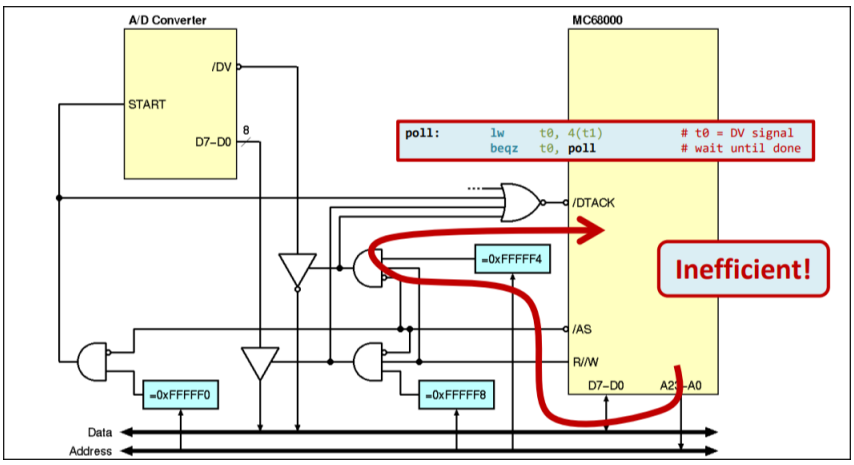
- Maybe the **stack cannot be used** (e.g., the exception results from a stack overflow)
- Maybe the **exception handler cannot be interrupted** (e.g., the handler uses static locations to save data including `mscratch`) and is therefore a nonreentrannt procedure)
- Maybe the **system cannot withstand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



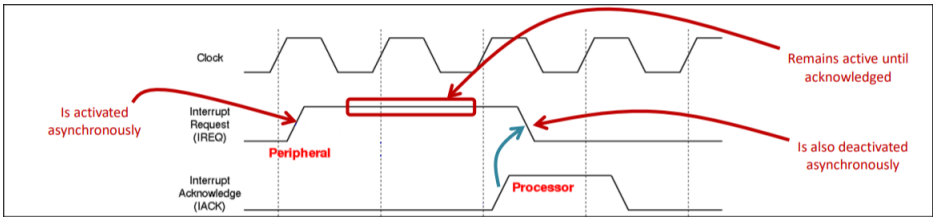
However this is pretty ineficient, the processor has to always check weither the signal is active or not instead of doing real work.
The goal here is to improve the interface to the A/D converter so that:

- 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.

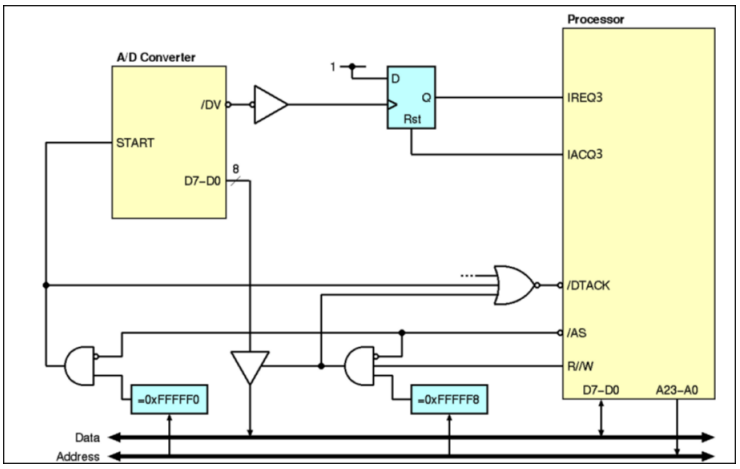
- The result of the conversion can be ready by the processor at address `oxFFFFF8`

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**

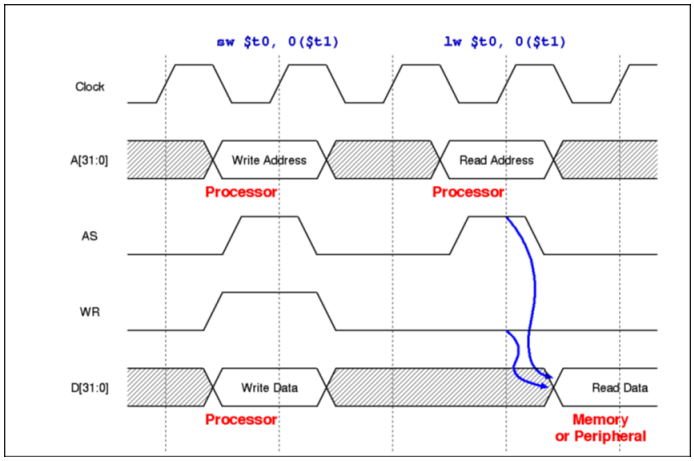
Today's lecture is an example of what we did the previous weeks.

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".

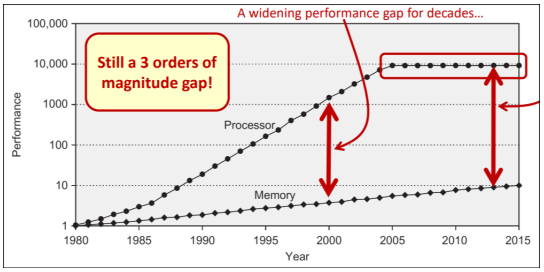
Chapter 4

Memory Hierarchy

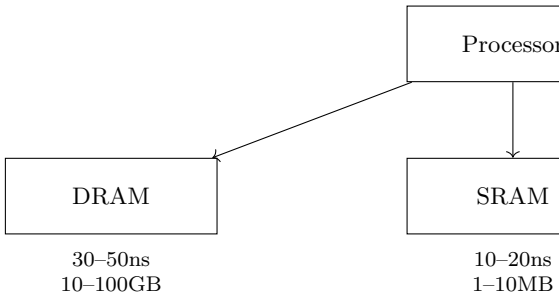
lectureDate2025-10-17 lectureDate — **Cours 12** What we want now is to go back to memory and spent quite a lot of time on this topic. What we will change here is that we will care about the performance of the memory we are building.

4.1 Caches

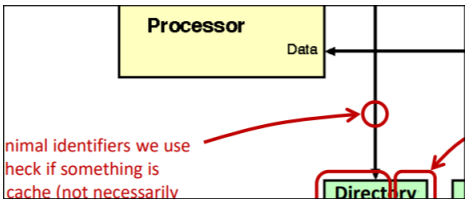
The question we have is what is the problem with memory:



Processor has been improved a **lot** in comparisaison of memory. What memory means here is DRAM.
The issue is that we want a big memory and a fast processor which cannot really work well together, we need to **so something** ourself.



What we can do is to use other type of memory which is **faster but expensive**



So the fact is: this is not only a technology issue even though we are not able to make `DRAM` faster. It's not even an issue with `SRAM` cost, it wouldn't be too costly to make `SRAM` bigger, the issue is that even though SRAM is pretty maleable, when expanding it, it becomes 2 to 3 degree of magnitude slower (in terms of clock cycles).

What memory to use?

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**should be in fast memory
- if variable `i` and `sum` are stored in memory, they are also **used often** and should 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 criteria 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 are likelyhood to be used in the future (Code are usually sequential, Data: array)

However, this is not perfect, this is onlyl 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:

Invisible to the programmer

- One could analyse data structures and program semantics to detect heabily used vairables/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 accesst 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 make 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 dirc