Theorie Computer Systemen 1

## Week 1 les 2: Computer Systemen 1: Intro, basispoorten, bits, bytes, codes

## Speed

**Bespreekt hoe relatief het begrip snelheid is wanneer we het over computers hebben.**

There is a theoretical limit to how fast they can go, but

engineers keep finding practical ways to get around the theories and make

machines that go faster and faster.

## ~~Language~~

~~Korte uitleg van de manier waarop het boek omgaat met begrippen in samenhang met computersystemen gebruikt worden.~~

## Just a Little Bit

**De betekenis van het begrip 'bit'.**

What do all of these things have in common? They are all places that contain athing that can be in one of two possible states. This is the definition of a bit.

A bit is some kind of a physical object that has a size and a location in space, and it has some quality about itself, that at any given time can be in one of two possible states, and maybe made to change back and forth between those two states.

## What the...?

**Eerste kennismaking met een poort en met een waarheidstabel.**

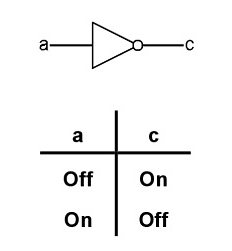
If you have been extra observant, you may have asked yourself this question: "when both inputs are off, the output is on, so.... how do you get electricity at the output if both inputs are off?" Well, that is an excellent question, and the excellent answer is that every one of these devices is also connected to power.

Like every appliance or table lamp in your house, where each has a plug with two pins, this device has a pair of wires, one of which is connected to a place where the electricity is always on, and the other is connected to a place where the electricity is always off.

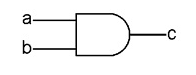
## Simple Variations

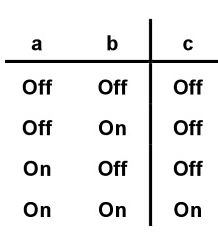
Introductie van enkele basispoorten:

* Not

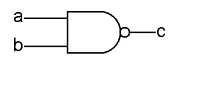


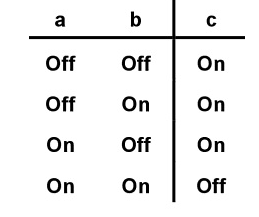
* AND





* NAND





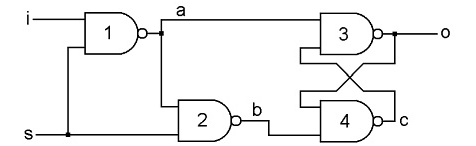
## Diagrams

Uitleg hoe de diagrammen in de rest van het boek gelezen moeten worden.

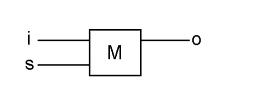
## Remember When

Werking van computergeheugen, introductie van de geheugendiagram.

MEMORY



Nadat set is ingedrukt is de output 1. De output blijft in deze state onveranderd, totdat set weer wordt ingedrukt. (flipflop)



## What Can We Do With A Bit?

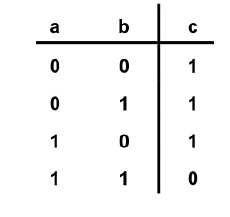
**Inleiding tot het begrijpen van het nut van een bit.**

Since a bit is actually nothing more than the electricity being on or off, the only actual, real thing we can do with a bit is to turn lights on or off, or toasters or whatever.

So there are many things that can be done with a bit. It can indicate true or false, go or stop. A single yes or no can be a major thing, as in the answer to "Will youmarry me?" or an everyday matter such as "Would you like fries with that?"

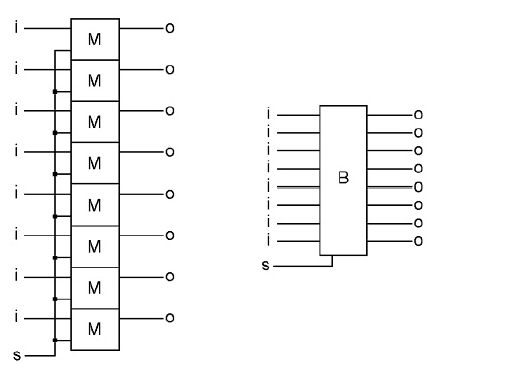
## A Rose by Any Other Name

Introductie van de waarden *0* en *1* voor uit respectievelijk aan.

Tada niet off/on maar 1 en 0 yay

## Eight Is Enough

Samenvoegen van 8 bits tot een byte.



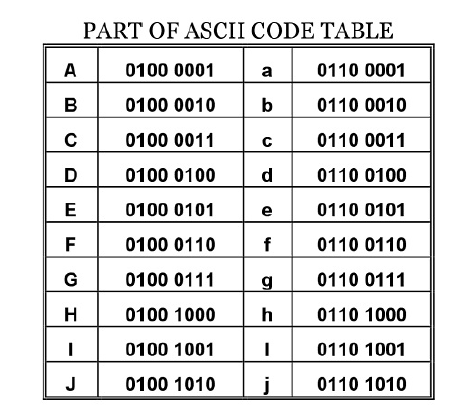
We are going to take eight bits, and call it a byte. Since a bit is a thing that has a location in space, that can be in one of two states, then a byte is a thing that has eight separate locations in space, each of which can be on or off, that are kept in the same order. The byte, taken as a whole, is a location in space that can be in any one of 256 states at any given time, and may be made to change its state over time.

Theorie Computer Systemen 2

Week 2 les 1: Computer Systemen 2: Registers, bus, poorten, RAM, getallen

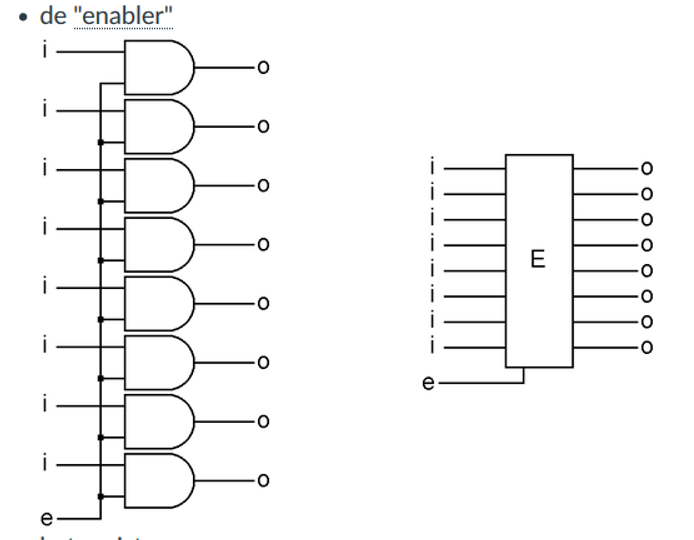
## Codes

De manier waarop we bytes een betekenis geven, bijvoorbeeld hoe met behulp van ASCII codering leesbare tekst gecodeerd kan worden als bytes.



**Back to the Byte**

Introductie van:

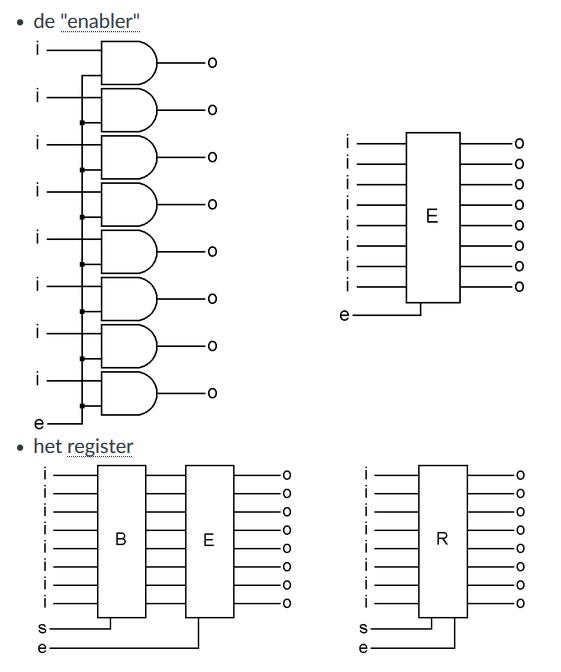


When 'e' is off, whatever comes into the Enabler

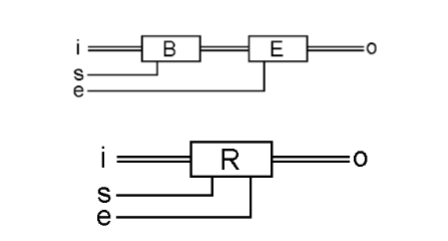
goes no further, because the other side of each AND gate is off, thus the outputs of those gates are all going to be off. When 'e' is on, the inputs go through the Enabler unchanged to the outputs, 'o.' Let op dat als ‘e’ uit is ook de waarde weer niks wordt (kortom: alle 8 tegelijk aan en uit)

Now we have a combination that can store eight bits. It captures them all at the same time, and it can either keep them to itself, or let them out for use

somewhere else. This combination of a Byte and an Enabler, has a name, it is called a Register, thus the 'R' in the drawing.



Kan verkort zo worden weergegeven:

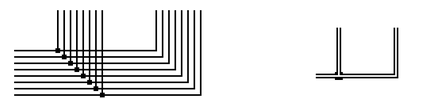


Now we have a combination that can store eight bits. It captures them all at the same time, and it can either keep them to itself, or let them out for use

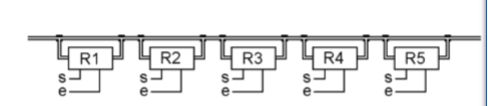
somewhere else. This combination of a Byte and an Enabler, has a name, it is called a Register, thus the 'R' in the drawing.

**The Magic Bus**

Introductie van de bus waarmee, analoog aan het rijdende exemplaar, meerdere bits tegelijk vervoerd kunnen worden. Daarna wordt uitgelegd hoe deze in diagrammen weergegeven kan worden.



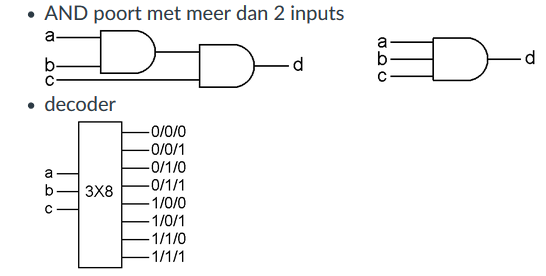
When there is a connection between two of these bundles of wires, one wire of each bundle is connected to one wire of the other bundle as shown in the diagram on the left. But we will simplify it, and just draw it like the diagram on the right.

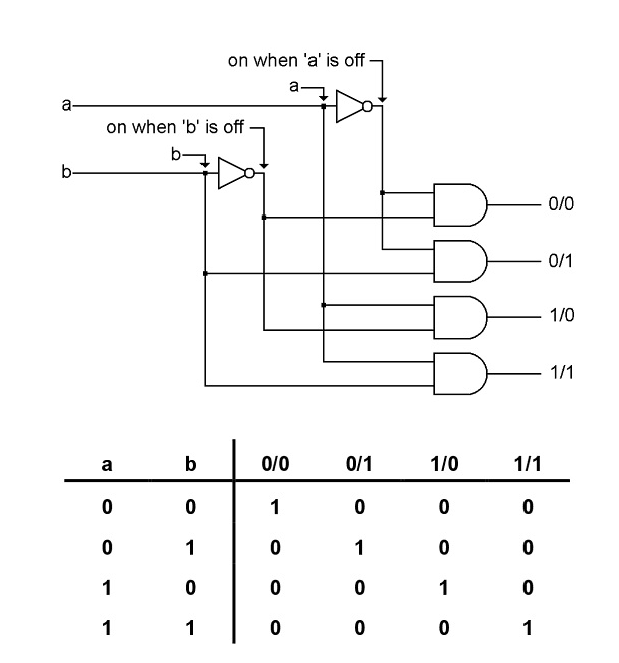


In the following example, we have a bus, and there are five registers, each ofmwhich has both its input and output connected to the same bus.

**More Gate Combinations**

Twee nieuwe poorten worden geïntroduceerd:

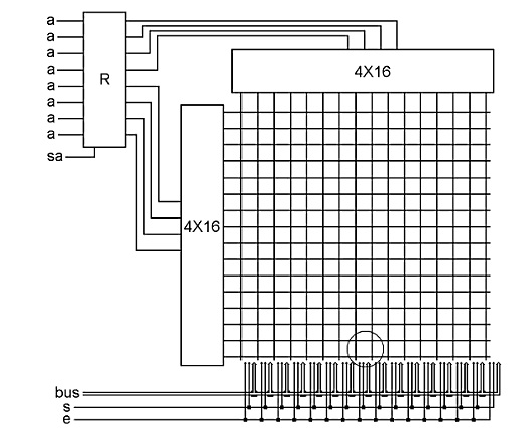




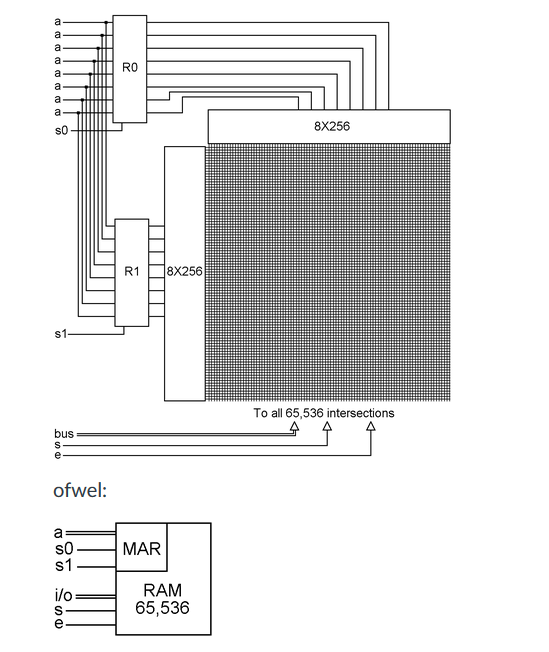
This combination is called a decoder. The name means that if you consider then four possible states of the two inputs as a code, then the output tells you which of the codes is currently on the input. Maybe it’s not a great name, but that’s what it meant to someone once, and the name stuck. This decoder has two inputs, which means that there can be four combinations of the states of the inputs, and there are four outputs, one corresponding to each of the possible input combinations.

**First Half of the Computer**

Alle tot dusverre behandelde onderdelen worden gebruikt om een halve computer samen te stellen: het computergeheugen (RAM).



Veder uitzoeken nu te lui



**Numbers**

Uitleg van de manier waarop een computer getallen codeert met behulp van bits: binaire getallen.

**Addresses**

Beschrijving van een belangrijk gebruik van binaire getallen: adressering van geheugen.

The number in MAR is considered to be a number somewhere

between o and 255, and thus each of the 256 RAM bytes can be considered to have an address.

So a computer address is just a number that causes a certain byte to be

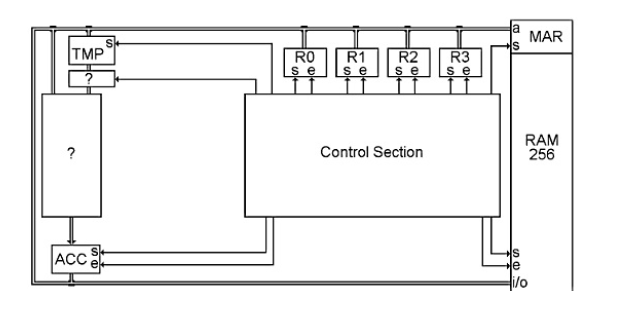
selected when that address is placed into the Memory Address Register.

Week 2 les 2: Computer Systemen 3: Byte operaties, logica, control unit

Theorie Computer Systemen 3

**The Other Half of the Computer**

Een korte inleiding tot de onderdelen die nodig zijn om een complete computer te maken nu registers en RAM gerealiseerd zijn.



Here are the beginnings of the CPU. The RAM is shown on the right, and the

bus makes a big loop between the two bus connections on the RAM. The CPU starts with six registers connected to the bus. These six registers are all of the places that the CPU will use to “process” bytes.

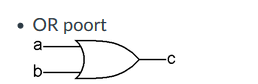
R0, R1, R2, and R3 are registers that are used as short-term storage for bytes that are needed in the CPU.

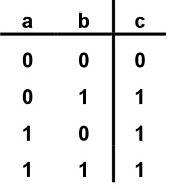
The register called 'TMP' means temporary. Its input comes from the bus, and its output goes downward to one and then the other of the question marked boxes. TMP has a 'set' bit, but no 'enable' bit because we never have a reason to turn its output off.

The last register is called the accumulator, or ACC for short. In a computer, it just means that it temporarily stores a byte that comes from that big question marked box. The output of ACC is then connected to our old friend, the bus, so it can be sent somewhere else as needed.

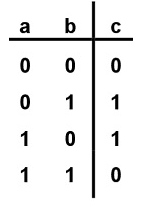
**MORE GATES**

Twee nieuwe poorten worden geïntroduceerd:

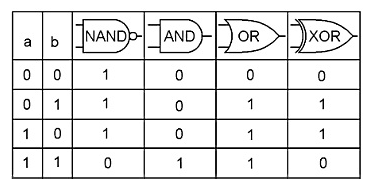








Poort samenvatting

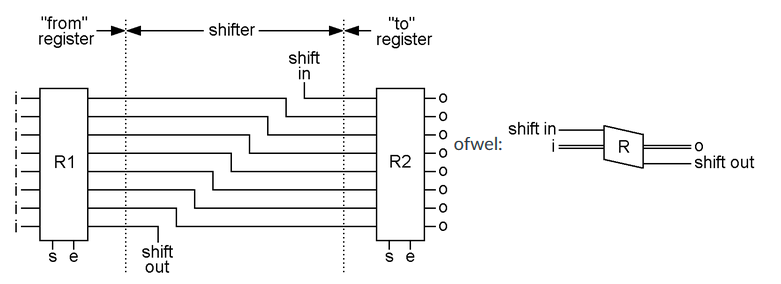


**Messing with Bytes**

Introductie tot de volgende hoofdstukken waarin operaties op bytes worden uitgewerkt.

**The Left and Right Shifters**

RIGHT

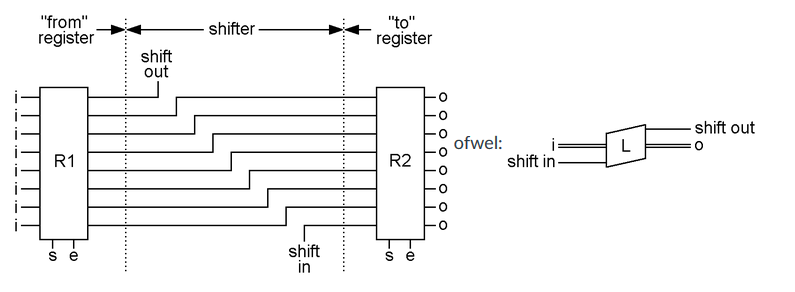


This shows two registers connected by a right shifter. The shifter is just the wires between the two registers. When the ‘e’ bit of R1 is turned on, and the ‘s’ bit of R2 is turned on and then off, all of the bits in R1 are copied into R2, but they are shifted over one position. The one at the bottom (shift out) can be connected to some other bit in the computer, but is often connected back to the one on the top (shift in) and when that is done, the rightmost bit wraps around to the leftmost bit at the other end of the byte.

A right shifter will change 0100 0010, to 0010 0001.

If ‘shift out’ is connected to ‘shift in,’ a right shift will change 0001 1001 to 1000 1100

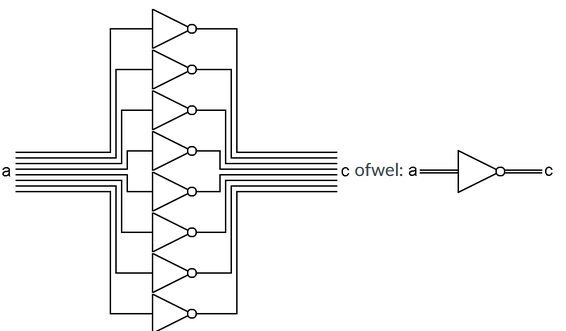
A left shifter will change 0100 0010 to 1000 0100. The left shifter is wired up like so:



Now of what use is this? The minds of programmers have come up with all sorts of things, but here is an interesting one. Assume that you are using the binary number code. You have the number 0000 0110 in R1. That comes out to the decimal number 6. Now shift that code left into R2. R2 will then be 0000 1100. This comes out to the decimal number 12. What do you know, we have just multiplied the number by 2. This is the basis of how multiplication is done. If you want to multiply something by ten, you just add a zero to the right side, effectively shifting each digit left one position.

**The NOTter (inverter)**

Werking, betekenis en notatie van een NOT poort voor bytes.

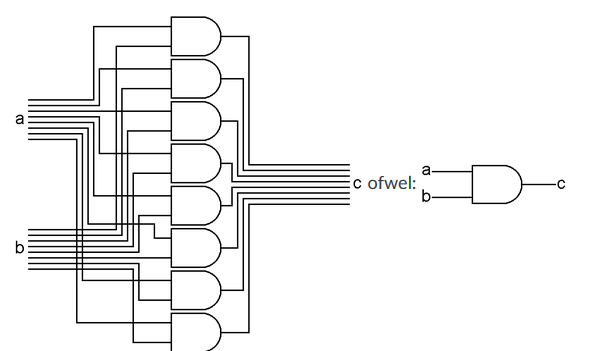


This device connects two registers with eight NOT gates. Each bit will be

changed to its opposite. If you start with 0110 1000, you will end up with 1001 0111.

**The ANDer**

Werking, betekenis en notatie van een AND poort voor bytes.



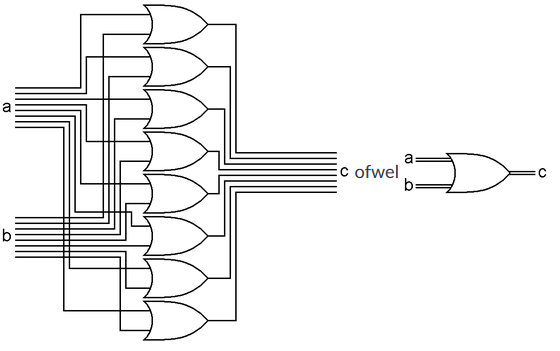
The ANDer takes two input bytes, and ANDs each bit of those two into a third byte. As you can see, the eight bits of the 'a' input bus are connected to the upper side of eight AND gates. The eight bits of the 'b' input bus are connected to the lower side of the same eight AND gates. The outputs of the eight AND gates form the bus output 'c' of this assembly. With this, we can AND two bytes together to form a third byte.

There are many uses for this. For example, you can make sure that an ASCII

letter code is uppercase. If you have the code for the letter 'e' in R0, 0110 0101, you could put the bit pattern 1101 1111 into R1 and then AND R1 and R0 and put the answer back into R0. All of the bits that were on in R0 will be copied back to R0 except for the third bit. Whether the third bit had been on or off before, it will now be off. R0 will now contain 0100 0101, the ASCII code for 'E.' This works for all ASCII letter codes because of the way ASCII is designed.

**The ORer**

Werking, betekenis en notatie van een OR poort voor bytes.

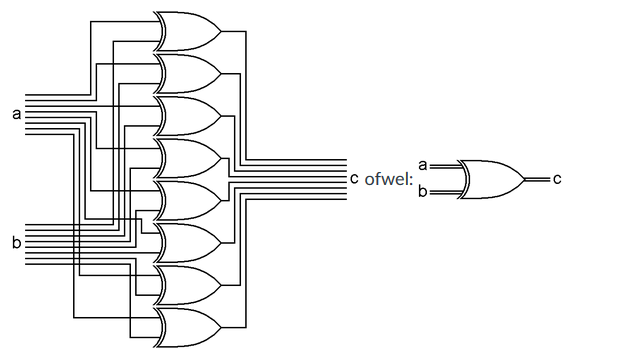


The ORer takes two input bytes, and ORs each bit of those two into a third byte. As you can see, the eight bits of the 'a' input bus are connected to the upper side of eight OR gates. The eight bits of the 'b' input bus are connected to the lower side of the same eight OR gates. The outputs of the eight OR gates are the bus output 'c' of this assembly. With this, we can OR two bytes together to form a third byte.

What is the use of this? There are many, but here is one of them. Say you have the ASCII code for the letter 'E' in R0, 0100 0101. If you want to make this letter be lowercase, you could put the bit pattern 0010 0000 into R1 and then OR R0 and R1 and put the answer back into R0. What will happen? All of the bits of R0 will be copied back into R0 as they were except the third bit will now be on no matter what it had been. R0 will now be 0110 0101, the ASCII code for 'e.' This will work no matter what ASCII letter code was in R0 because of the way ASCII was designed.

**The Exclusive ORer**

Werking, betekenis en notatie van een XOR poort voor bytes.

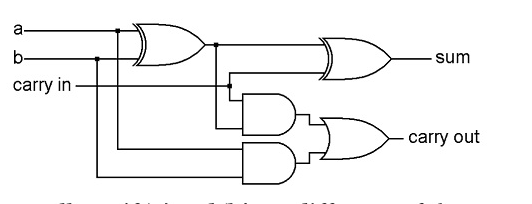


The XORer takes two input bytes, and XORs each bit of those two into a third byte. As you can see, the eight bits of the 'a' input bus are connected to the upper side of eight XOR gates. The eight bits of the 'b' input bus are connected to the lower side of the same eight XOR gates. The outputs of the eight XOR gates are the bus output 'c' of this assembly. With this, we can XOR two bytes together to form a third byte.

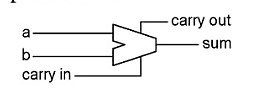
What is the use of this? Again, imaginative people have come up with many of uses. But here is one of them. Say you have one code in R1 and another code in R2. You want to find out if those two codes are the same. So you XOR R1 and R2 into R1. If R1 and R2 contained the same patterns, then R1 will now be all zeros. It doesn't matter what pattern of 0s and 1s was in R1, if R2 contained the same pattern, after an XOR, R1 will be all zeros.

**The Adder**

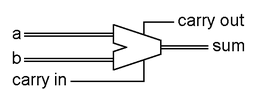
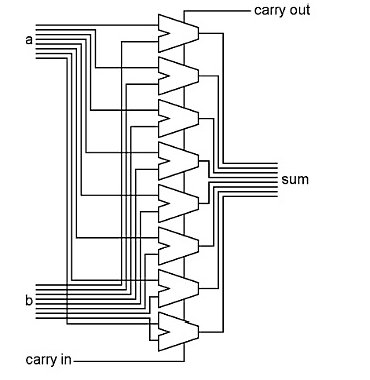
Introductie van binair optellen en 'carry' bit gevolgd door werking, betekenis en notatie van een poort die bytes bij elkaar kan optellen.



The left XOR tells us if ‘a’ and ‘b’ are different. If they are, and ‘carry in’ is off, or if ‘a’ and ‘b’ are the same and ‘carry in’ is on, then the right XOR will

generate 1 as the sum for the current column. The lower AND gate will turn on ‘carry out’ if both inputs are on. The upper AND gate will turn on ‘carry out’ if ‘carry in’ and one of the inputs are on. This is the simplicity of how computers do addition! Now that we see that it works, we can make a simpler picture of it: 

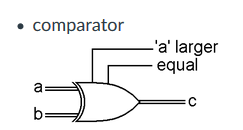
To make an adder that adds two bytes together, we need a one bit adder for each bit of the bytes, with the carry output of each bit connected to the carry input of the next. Notice that every bit has a carry in, even the first bit (the right column.) This is used when we want to add numbers that can be larger than 255.



The carry output bit of the leftmost (top) column will turn on if the sum of the two numbers is greater than 255, and this bit will be used elsewhere in the computer. This is how computers do addition, just five gates per bit, and the computer can do arithmetic (rekenkunddig)!

## The Comparator and Zero

Werking, betekenis en notatie van een poort waarmee bytes met elkaar kunnen worden vergeleken en een poort die een byte vergelijkt met 0:



All of the things we have described above take one or two bytes as input, and generate one byte of output. The shifters and the adder also generate one extra bit of output that is related to their output byte. The comparator only generates two bits of output, not a whole byte.

What we want the comparator to do, is to find out whether the two bytes on the input bus are exactly equal, and if not, whether the one on the 'a' bus is larger according to the binary number system.

**The equal thing is pretty simple. XOR gates turn off when the inputs are the same, so if all of the XOR gates are off, then the inputs are equal.**

To determine the larger of two binary numbers is a little trickier. **You have to start with the two top bits**, **and if one is on and the other is off, then the one that is on is the larger number. If they are the same, then you have to check the next lower pair of bits etc., until you find a pair where they are different. But once you do find a pair that are different, you don't want to check any more bits. For example, 0010 0000 (32) is larger than 0001 1111 (31.)** The first two bits are the same in both bytes. The third bit is on in the first byte and off in the second, and therefore the first byte is larger. Although the rest of the bits are on in the second byte, their total is less than the one bit that is on in the first byte.

There is one more thing that we are going to need in our computer that gives us another bit of information. This is a simple gate combination that takes a whole byte as input, and generates only one bit as output. The output bit turns on when all of the bits in the byte are off. In other words, the output bit tells us when the contents of the byte is all zeros.

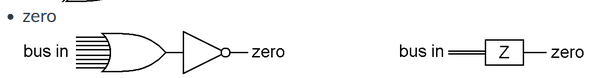
It is simply an eight input OR gate and a NOT gate. When any of the inputs to the OR gate are on, its output will be on, and the output of the NOT gate will be off. Only when all eight inputs of the OR are off, and its output is therefore off, will the output of the NOT gate be on. The simpler bus representation is shown on the right.

10001100

01110000

11111100

00000000

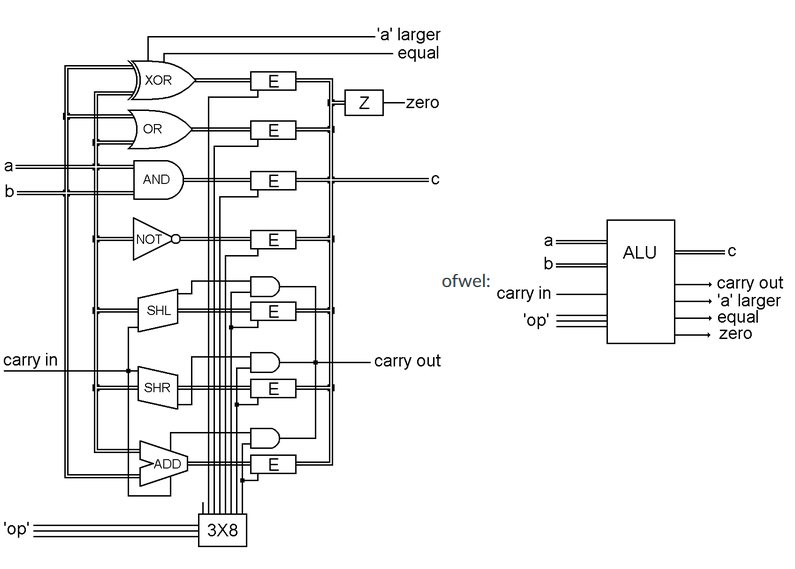


## Logic

Korte introductie tot de achtergrond van de termen "logica" en "rekenkunde" in de context van computer systemen.

## The Arithmetic and Logic Unit

Alle logische en rekenkundige poorten worden samengevoegd tot de "Arithmetic and Logic Unit" ofwel ALU.



All seven devices are connected to input 'a,' the devices that have two inputs are also connected to input 'b.'

All seven devices are connected to the inputs at all times, but each output is attached to one of those enablers.

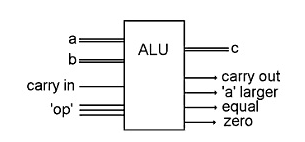
The wires that turn the enablers on, are connected to the outputs of a decoder, thus only one enabler can be on at a given time.

Seven of the decoder's outputs enable a single device to continue on to the common output, 'c.' Seven of the decoder's outputs enable a single device

to continue on to the common output, 'c.' The eighth output of the decoder is used when you don't want to select any device at all. (zie leeg kabbeltje)

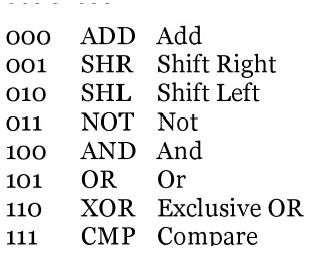
The three input wires to the decoder are labeled 'op,' because they choose the desired 'operation.' The one little complication here is the carry bits from the adder, and the ‘shift in’ and ‘shift out’ bits from the shifters. These are used in very similar ways, and so from here on out we will refer to all of them as carry bits.

The adder and both shifters take carry as an input, and generate carry as an output. So the three carry inputs are connected to a single ALU input, and one of the three outputs is selected along with the bus output of its device.

What do we have here? It is a box that has two bus inputs, one bus output and four other bits in and four other bits out. Three of the input bits select which “operation” will take place between the input and output buses. Again, now that we know what’s in it and how it works, we don’t need to show all of its parts. Here is a simplified way to draw it: 

Notice that the three single bit inputs labeled “op,” above, can have eight

different combinations. Seven of those combinations select one of the devices described previously. The eighth combination does not select any output byte, but the ‘a larger’ and ‘equal’ bits still work, as they do at all times, so this is the code to choose if you only want to do a comparison.

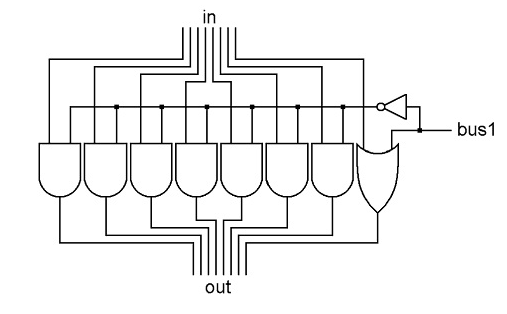


The combination of bits at ‘op’ mean something. That sounds like another code. Yes, here is a three-bit code that we will make use of soon.

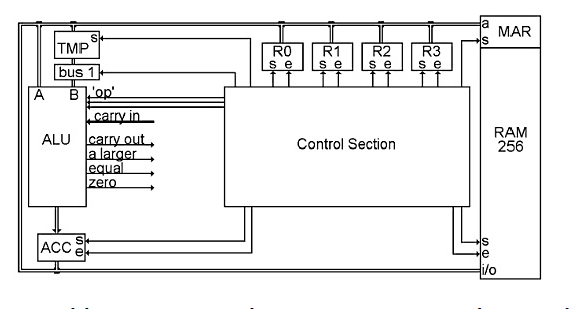
**More of the Processor**

Introductie van een component dat de waarde '1' op een bus kan zetten en de samenhang tussen alle tot nu toe behandelde onderdelen van de processor: alleen de uitwerking van de aansturing ontbreekt.

It is very similar to an enabler. Seven of the bits go through AND gates, and one of them goes through an OR gate. The single bit input determines what happens when a byte tries to pass through this device. When the 'bus 1' bit is off, all of the bits of the input bus pass through to the output bus unchanged. When the 'bus 1' bit is on, the input byte is ignored and the output byte will be 0000 0001, which is the number l in binary. We will call this device a 'bus 1' because it will place the number 1 on a bus when we need it.



Now we can put this 'bus 1' and the ALU into the CPU. We will change where the wires go in and out of the ALU so it fits our diagram better. The bus inputs come in the top, the bus output comes out the bottom and all of the input and output bits are on the right.



The output of the ALU is connected to ACC. ACC receives, and temporarily

stores, the result of the most recent ALU operation. The output of ACC is then connected to the bus, so its contents can be sent somewhere else as needed.

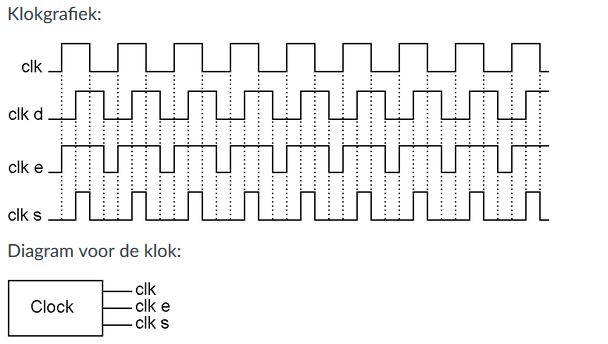
When we want to do a one input ALU operation, we have to set the three ‘op’ bits of the ALU to the desired operation, enable the register we want onto the bus, and set the answer into ACC.

For a two input ALU operation, there are two steps. First we enable one of the registers onto the bus and set it into TMP. Then we enable the second register onto the bus, choose the ALU operation, and set the answer into ACC.

There is only one thing missing here, and that has to do with all of these control bits on the registers, ALU and RAM. The RAM has three control bits, one to set MAR, one to set the currently selected byte in, one to enable the currently selected byte out. Each of the registers, R0, R1, R2, R3 and ACC have a set and an enable bit, TMP only has a set bit, bus 1 has a control bit, and the ALU has those three ‘op’ bits that select the desired operation

**The Clock**

Betekenis, werking en notatie van de 'klok' in een computer systeem.



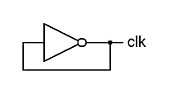
When something repeats some action regularly, one of those actions,

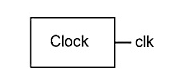
individually, is called a cycle. The graph above shows about eight cycles.

500 Hz means 500 times per second. For faster speeds we run into those

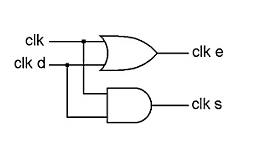
ancient languages again, and one thousand times per second is called a kilohertz or kHz for short. Going on and off a million times per second is called a megahertz, or mHz for short, and a billion times is called a gigahertz, or gHz for short.

This one special bitturns on and off all by itself. But there is nothing mysterious about it, it just goeson and off very regularly and very quickly. This special bit is built very simply,like this:



Simplify 

This bit has a name. It is called the clock.

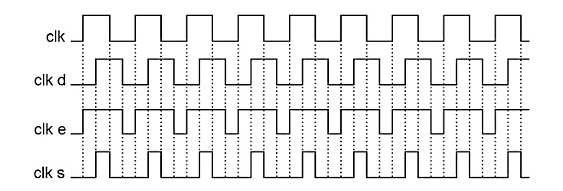


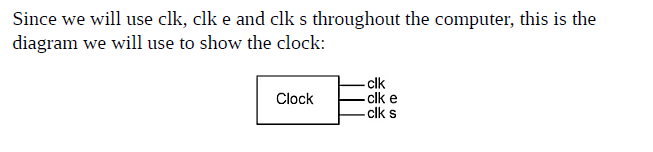
The graph of the inputs and outputs of the AND and OR gates is shown here. They are both still going on and off regularly, but one of them is on for longer than it is off, and the other one is off for longer than it is on.

Clk\_d = delay

Clk\_e = enable

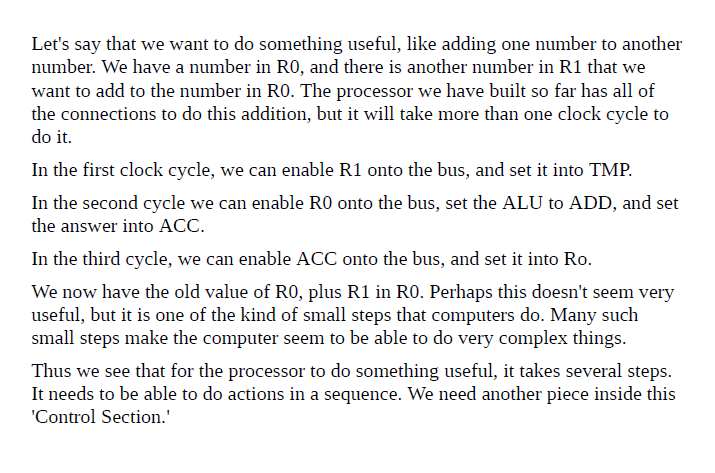
Cls\_s = set





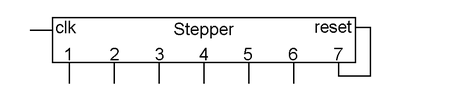
**Doing Something Useful**

Uitleg van de stappen die een processor doet om een zinvolle operatie uit te voeren.



**Step by Step**

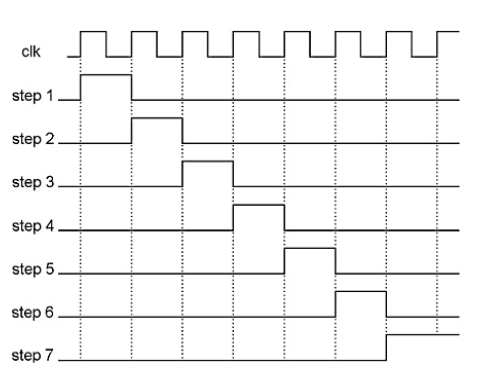
Betekenis, werking en notatie van de 'stepper' die nodig is om een zinvolle operatie geautomatiseerd uit te laten voeren.



One is called 'clk,' because this is where we connect a bit that

is going on and off, such as our original clock bit. The other input is called

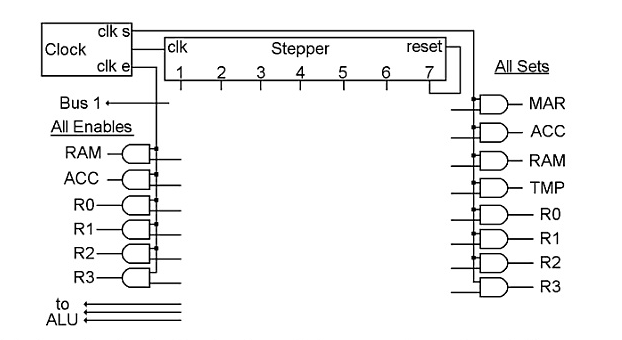
'reset,' which is used to return the stepper back to step one. For outputs, it has a number of bits, each of which will come on for one complete clock cycle, and then turn off, one after the other. The output labeled 'Step 1' turns on for one clock cycle, then 'Step 2' for the next clock cycle, etc. A stepper can be built to have as many steps as needed for any particular task you want to do. In the case of this computer that we are building, seven steps are sufficient. When the last step (7) turns on, it stays on, and the stepper doesn't do anything else until the reset bit is turned on briefly, at which time the steps start over again beginning with 'Step 1.'



**Everything's Under Control**

Een overzicht over de wijze waarop klok en stepper onderdeel uitmaken van de aansturing van een processor.

'Control Section' that we know nothing about yet. Now it is time to look inside that box.



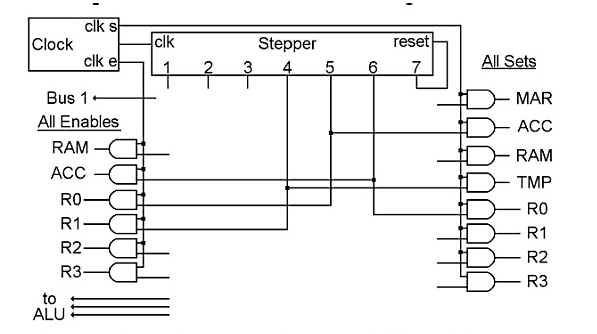
This drawing is the beginning of the control section of the computer. At the topnare the clock and the stepper. Then all of the control bits from the registers and RAM have been brought here together in one place, with all of the 'enable' bits on the left, and all of the 'set' bits on the right.

Then we have connected the output of an AND gate to each 'enable' and each 'set' bit. One input of each AND gate is connected to either 'clk e' for the 'enables' on the left, or 'clk s' for the 'sets' on the right. Thus

To recap, this is a stepper. It has two inputs: a clock and a reset. For outputs, it has a number of bits, each of which will come on for one clock cycle.

**Doing Something Useful, Revisited**

Uitleg op welke wijze de stepper aangesloten kan worden om een zinvolle operatie mogelijk te maken.

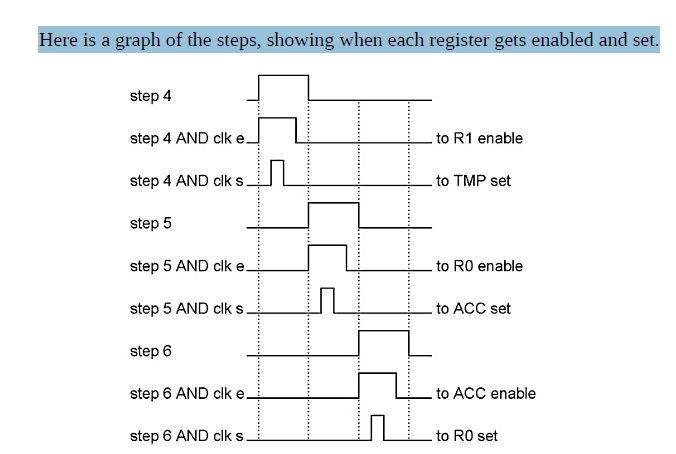


All we have to do to 'do something useful,' like adding R1 to R0, is to connect a few wires in the middle, as shown in this diagram with steps four, five and six.Each step causes something to happen to some of the parts that are shown in the CPU diagram. Each step is connected to one 'enable' on the left, and one 'set' on the right, and therefore causes one part to connect its output to the bus, and another part to save what now appears at its input. Step four is wired to R1 'enable' and TMP 'set.' Step five is wired to R0 'enable,' and ACC 'set.' The ALU 'op' bits do not need any connections since the 'op' code for ADD is 000. Step six is wired to ACC 'enable' and R0 'set.'

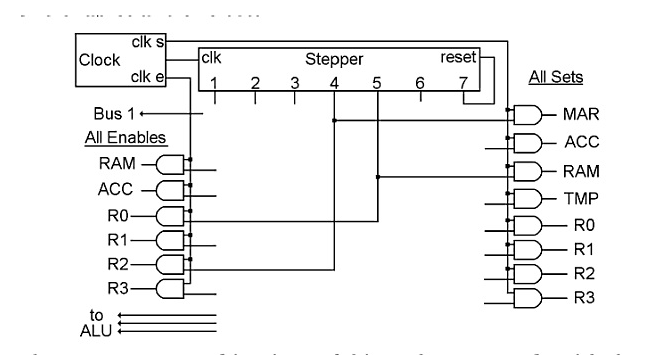
During step four, R1 is enabled and TMP is set. The contents of R1 travel across the bus (in the CPU diagram) and are captured by TMP. During step five, R0 is enabled and ACC is set. If we wanted to do something

other than ADD, this is the step where we would turn on the appropriate ALU 'op' code bits. During step six, ACC is enabled and R0 is set.

In step seven, the stepper is reset to step 1, where the process repeats. Of course it is not very useful to just do this addition over and over again, even if you start out with the number 1 in both R0 and R1, R0 will get up to 255 pretty quickly.



Perhaps now that we have added R1 to R0, we want to store that new number to a particular address in RAM, and R2 happens to have that address in it. Again, our processor has all of the connections necessary to do this, and again it will take more than one clock cycle to do it. In step 4, we can move R2 across the bus to MAR. In step 5 we can move R0 across the bus to RAM. That's all that is needed, just two clock cycles and we're done.



# Week 3 les – 1 Theorie Computer Systemen 4

**What's Next?**

Inleiding tot het idee achter code voor het uitvoeren van een taak (ofwel een algoritme of programma).

**The First Great Invention**

Uitleg van wat nodig is om de processor *verschillende* taken uit te laten voeren: opdrachten moeten uit het het RAM geheugen worden opgehaald, met behulp van drie onderdelen:

* Instruction Register (IR)
* Instruction Address Register (IAR)
* Verbindingen met de [stepper](https://canvas.hu.nl/courses/890/pages/theorie-computer-systemen-3#stepper) die ervoor zorgen dat een opdracht vanuit het RAM in het IR gezet kan worden, dat het adres in het IAR met 1 verhoogd wordt tot slot dat de opdracht in het IR wordt uitgevoerd

Hierbij wordt het begrip "programma" geïntroduceerd.

Deze stappen hangen nauw samen met de in de presentatie (maar niet in het boek) behandelde "Von Neumann cyclus": Fetch-Decode-Execute-Store, alleen het 'fetch' deel wordt in dit hoofdstuk verder uitgewerkt (stappen 1, 2 en 3 van de stepper).

The invention is that we will have a series of instructions in RAM that will tell the CPU what to do. We need three things to make this work.

**Instruction register**

This register will be called the "Instruction Register," or "IR" for short.

The bits from this register will "instruct" the CPU what to do. The IR gets its

input from the bus, and its output goes into the control section of the CPU where the bits select one of several possible operations.

**Instruction Address Register**

The second part of the invention is another register in the CPU called the

"Instruction Address Register," or "IAR" for short. This register has its input and output connected to the bus just like the general purpose registers, but this one only has one purpose, and that is to store the RAM address of the next instruction that we want to move into the IR. If the IAR contains 0000 1010 (10 decimal,) then the next instruction that will be moved to the IR is the byte residing at RAM address ten.

**Wiring instruction**

The third part of the invention is some wiring in the control section that uses the stepper to move the desired "instruction" from RAM to the IR, add 1 to the address in the IAR and do the action called for by the instruction that has been put in the IR. When that instruction is complete, the stepper starts over again, but now the IAR has had 1 added to it, so when it gets that instruction from RAM, it will be a different instruction that was located at the following RAM address.

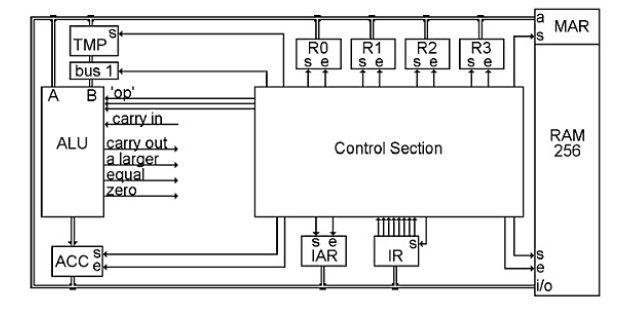
All we have to do is to place a series of bytes in RAM that represent

a series of things that we want to do, one after another.This series of bytes residing in RAM that the CPU is going to make use of is

called a "program."

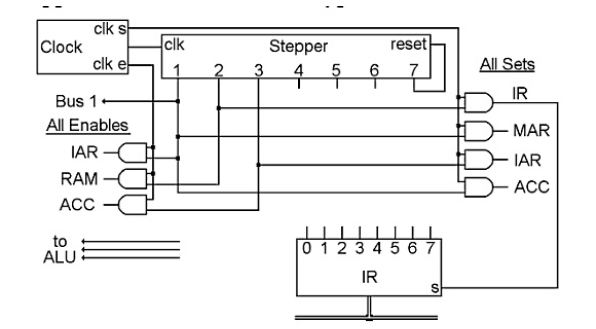
The stepper in this computer has seven steps. The purpose of step 7 is only to reset the stepper back to step 1. So there are six steps during which the CPU does small things. j

Here is the picture of the CPU with the two new registers added to it. There they are under the Control Section, connected to the bus. The IAR has a 'set' and 'enable,' the IR only has a 'set,'just like TMP and MAR because their outputs are not connected to the bus, so we never need to turn them off.



Below is the wiring within the Control Section that does the 'fetch' part of the instruction cycle. It uses the first three steps of the stepper and is the same for all types of instructions

**Fetch**



The stepper's first three steps are shown here, and result in 'fetching' the next 'instruction' from RAM. Then the rest of the steps 'execute' the 'instruction.' Exactly what will be done in steps 4, 5 and 6, is determined by the contents of the instruction that was fetched. Then the stepper starts over, fetches the next instruction, and executes it.

Step 1 is the most complicated because we actually accomplish two things at themsame time. The main thing we want to do is to get the address in IAR over to MAR. This is the address of the next instruction that we want to fetch from RAM. If you look at the wire coming out of step 1 of the stepper, you can see that two of the places it is connected to are the 'enable' of IAR and the 'set' of MAR. Thus, the contents of IAR will be placed on the bus during 'clk e' and set into MAR during 'clk s.' Sometime during the instruction cycle, we need to add 1 to the value in IAR, and since IAR is already on the bus, we might as well do it now. If we don't send anything to the ALU's 'op' bits, they will all be zero, and since 000 is the code for ADD, the ALU will be doing an ADD operation on whatever is on its two inputs, and presenting the answer to ACC. One input comes from the bus, which has IAR on it during this time. If we also turn on the 'bus 1' bit during step 1, the other input to the ALU will be a byte with the binary value of 1. If we turn on the 'set' of ACC during 'clk s,'we will capture the sum of IAR plus 1 in ACC. This just happens to be the address of

Step 2 enables the currently selected byte in RAM onto the bus, and sets it into IR. This is the instruction that we will 'execute' in steps 4, 5 and 6 of this

instruction cycle. In the diagram, you can see that the wire coming from step 2 is connected to the 'enable' of RAM and the 'set' of IR.

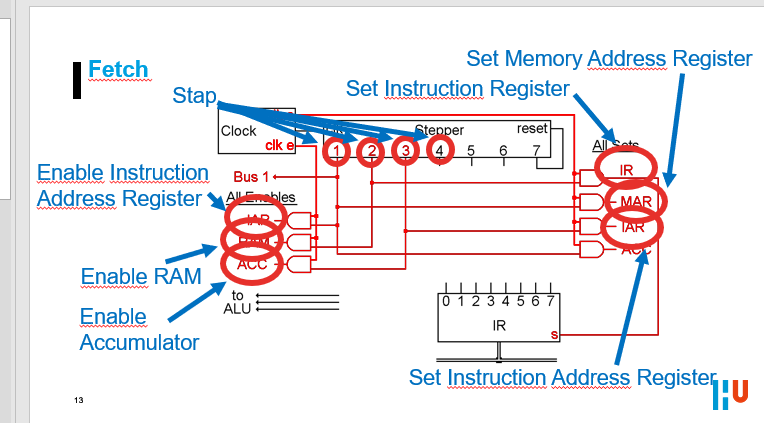
In step 3, we need to finish updating IAR. We added 1 to it in step 1, but the

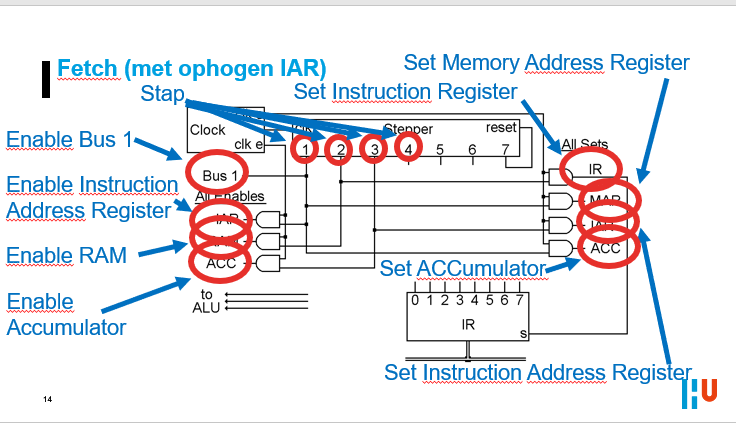
answer is still in ACC. It needs to be moved to IAR before the beginning of the next instruction cycle. So you can see the wire coming out of step 3 is connected to 'enable' of ACC and 'set' of IAR.

By the time we get to step 4, the instruction has already been moved from RAM to IR, and now steps 4, 5 and 6 can then do whatever is called for by the contents of IR. When that operation is done and the stepper is reset, the sequence will start over again, but now IAR has had 1 added to it, so the instruction at the next RAM address will be fetched and executed.

TOT IN WELKE DETAILS MOETEN WE DIT Weten VOOR HET TENTAMEN?

LES:





**Instructions**

Introductie van het begrip 'Instruction Code', de codering waarmee opdrachten als binaire waarden in RAM opgeslagen kunnen worden.

**Instruction Code**

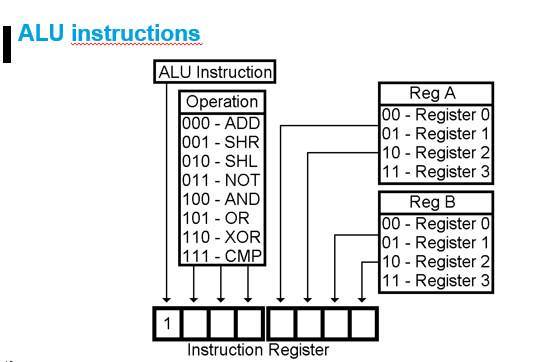
We now have this new register, called the Instruction Register, which contains a byte that is going to tell the Control Section what to do. The patterns that are put into this register have a meaning. Sounds like another code, and indeed, it is. This code will be called the "Instruction Code."

**The Arithmetic or Logic Instruction**

Uitwerking van het eerste van drie typen opdrachten die de processor kan uitvoeren: rekenkundige/logische opdrachten, die afgehandeld worden door de [ALU](https://canvas.hu.nl/courses/890/pages/theorie-computer-systemen-3#ALU).

Hierbij wordt zowel ingegaan op de codering van deze opdrachten (Von Neumann cyclus: decode) als op de wijze waarop deze gerealiseerd (Von Neumann cyclus: execute en store) worden door verbindingen tussen onderdelen van de processor (stappen 4, 5 en 6 van de stepper).

Tot slot wordt een (vergeleken met binaire codering) voor mensen beter leesbare codering geïntroduceerd, benoemd als 'computer language' maar in de praktijk beter herkenbaar als 'assembly language'.



Here is the Instruction Code for the ALU instruction. If the first bit in the

Instruction Register is a 1, then this is an ALU instruction. That’s the simplicity of it. If the first bit is on, then the next three bits in the instruction get sent to the ALU to tell it what to do, the next two bits choose one of the registers that will be used, and the last two bits choose the other register that will be used.

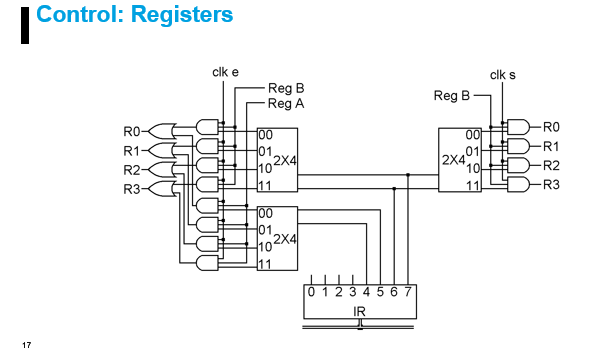
If you choose a one input operation, such as SHL, SHR or NOT, the byte will

come from the Reg A, go through the ALU, and the answer will be placed in the Reg B.

For two input operations, Reg A and Reg B will be sent to the ALU, and the

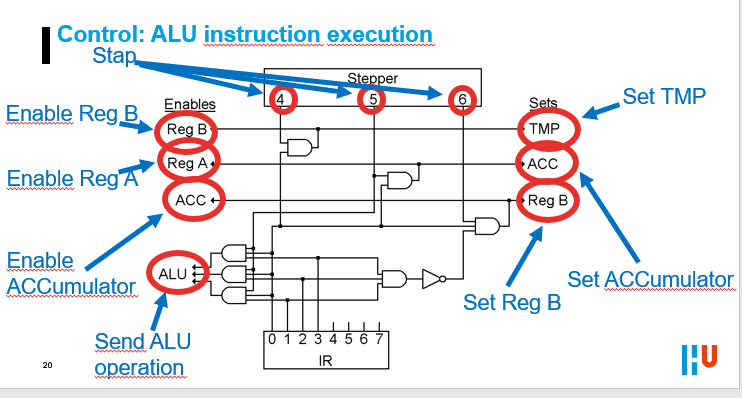
answer will be sent to Reg B. So whatever was in Reg B, which was one of the inputs to the operation, will be replaced by the answer.

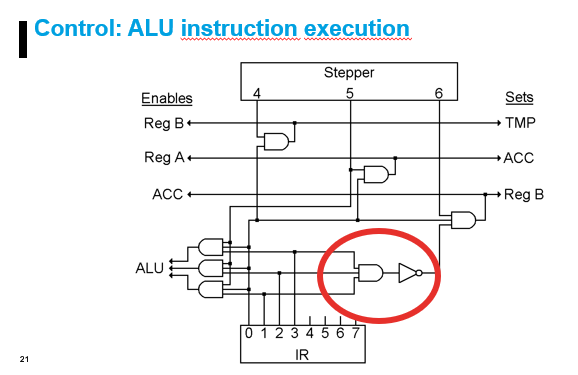
The CMP operation takes two inputs and compares them to see if they are equal, and if not, if the first one is larger. But the CMP operation does not store its output byte. It does not replace the contents of either input byte.



Look at the right side first. When we want to set a general-purpose register, we connect the proper step to this wire that we will call 'Reg B.' As you can see, 'elk s' is connected to all four AND gates. 'Reg B' is also connected to all four AND gates. But these four AND gates each have three inputs. The third input to each AND gate comes from a 2x4 decoder. You remember that one and only one output of a decoder is on at any given time, so only one register will actually be selected to have its 'set' bit turned on. The input to the decoder comes from the last two bits of the IR, so they determine which one register will be set by this wire labeled 'Reg B.' If you look back at the chart of the bits of the ALU Instruction Code, it shows that the last two bits of the instruction are what determine which register you want to use for Reg B.

The left side of the picture is very much like the right side, except that there are two of everything. Remember that in an ALU instruction such as ADD, we need to enable two registers, one at a time, for the inputs to the ALU. The last two bits of the instruction are also used for 'Reg B' on the left, and you can see that 'elk e,' 'Reg B' and a decoder are used to enable one register during its proper step. Bits 4 and 5 of the IR are used to enable 'Reg A' during its proper step, using a separate decoder and a wire called 'Reg A' The outputs of these two structures are ORed together before going to the actual register enable bits. We will never select 'Reg A' and 'Reg B' at the same time.





There is just one special situation in an ALU instruction, and that is when the

operation is CMP, code 111. For a compare operation, we do not want to store any results back into 'Reg B.' Therefore, there is a three input AND gate connected to IR bits 1, 2 and 3, which is then connected to a NOT gate, and then to a third input on the AND gate that does step 6 of the ALU instruction. So when the operation is 111, the first AND will come on, the NOT will go off, and the output of the Step 6 AND gate will not turn on

Soms kan het zijn dat

Nog even vragen waarom hij dit ook alweer aankaarten als speciaal.

Meer over inlezen blz 116

**Assambly code**

We are going to invent one more thing here, and that is a shorthand way of

writing CPU instructions on a piece of paper.

We are going to invent one more thing here, and that is a shorthand way of

writing CPU instructions on a piece of paper. In the Instruction Code, 1000 1011means "Add R2 to R3," but it takes a lot of practice for a person to look at 1000 1011 and immediately think of addition and registers. It also would take a lot of memorization to think of it the other way around, that is, if you wanted to XOR two registers, what is the Instruction Code for XOR? It would be easier to write something like ADD R2,R3 or XOR R1,R1.

## The Load and Store Instructions

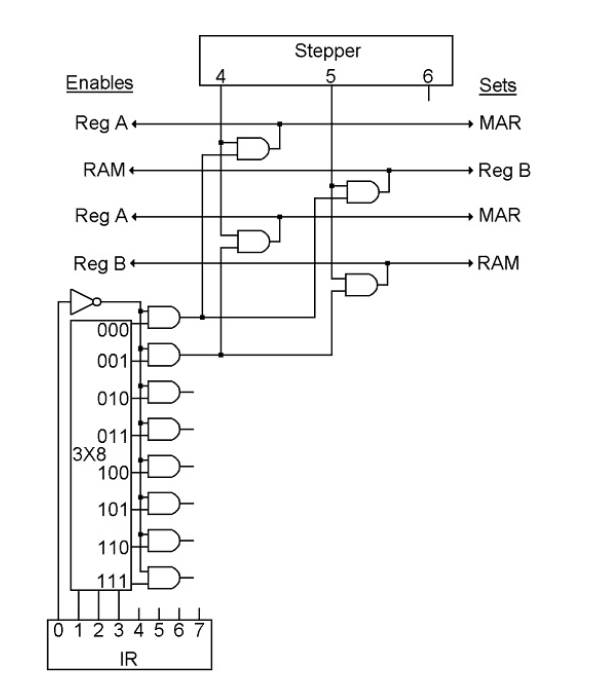
Uitleg en implementatie van de Load en Store opdrachten waarmee data gekopieerd kan worden van RAM naar een register en vice versa. Daarbij wordt meteen de basis gelegd voor de resterende instructies (alles wat niet door de ALU wordt afgehandeld).

The Load and Store instructions are pretty simple. They move a byte between RAM and a register. They are very similar to each other so we will cover both of them in one chapter.

We'll get to the details of these instructions in a moment, but first we need to have something that tells us when we have a Load or Store instruction in the Instruction Register. With the ALU instruction, all we needed to know was that bit o was on. The code for every other type of instruction begins with bit o off, so if we connect a NOT gate to bit o, when that NOT gate turns on, that tells us that we have some other type of instruction. In this computer, there are eight types of instructions that are not ALU instructions, so when bit o is off, we will use the next three bits of the IR to tell us exactly which type of instruction we

have.

In the diagram below, you can see IR bits 1, 2 and 3 going into a decoder which has eight AND gates on its outputs. IR bit o has a NOT gate which goes to the other side of those eight AND gates. This decoder is used for the rest of the instructions that our computer will have

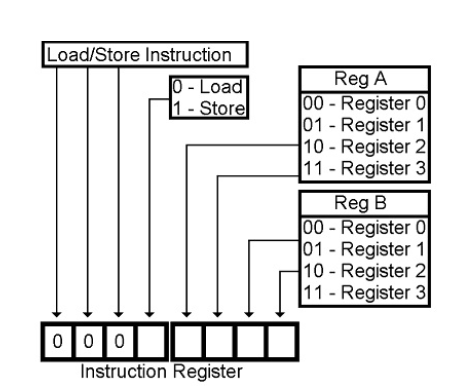


This chapter is about the instructions that use the first two outputs of the

decoder, the ones that come on when the IR starts with 0000 or 0001.

The first instruction moves a byte from RAM to a register, this is called the

"Load" instruction. The other one does the same in reverse, it moves a byte from a register to RAM, and is called the "Store" instruction.



The Instruction Code for the Load instruction is 0000, and for the Store

instruction is 0001.

Step 4 is the same for both instructions. One of the registers is selected by IR bits 4 and 5 and is enabled onto bus. The bus is then set into MAR, thus

selecting one address in RAM. In step five, IR bits 6 and 7 select another one of the CPU registers. For the Load instruction, RAM is enabled onto the bus and the bus is set into the selected register. For the Store instruction, the selected register is enabled onto the bus and the bus is set into RAM.

Each of these instructions only need two steps to complete, step 6 will do

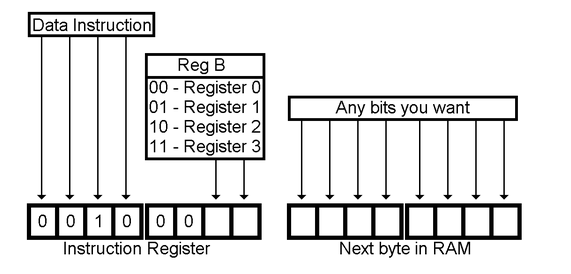
nothing.

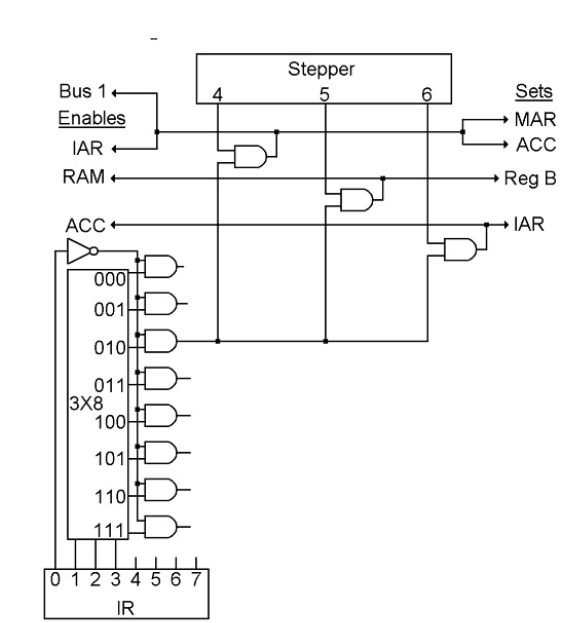
**The Data Instruction**

Uitleg en implementatie van de Data opdracht waarmee data naar een register kan worden gekopieerd vanuit *het deel van het RAM waar zich de opdrachten bevinden*. Op die manier kan een programma gebruik maken van waarden die in het programma zijn opgenomen.

All it does is load a byte from RAM into aRegister like the Load instruction, above. The thing that is different about it

though, is where in RAM it will get that byte.





Indien gewenst veder uitzoeken

**The Second Great Invention**

Uitleg van het concept van sprongen door het programma dat zich in het geheugen bevindt, en de implementatie van de onvoorwaardelijke sprong naar een adres in een register.

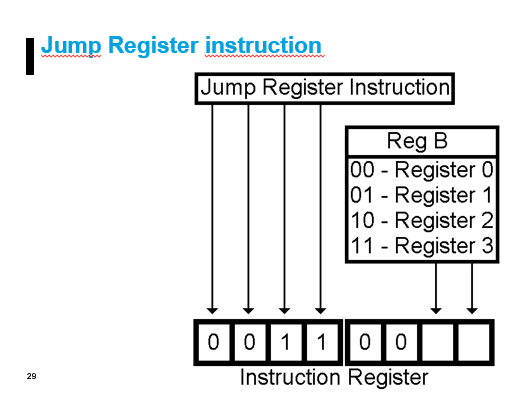
This new type of instruction is called a Jump instruction, and all it does is to

change the contents of the IAR, thus changing where in RAM the next, and

subsequent instructions will come from.

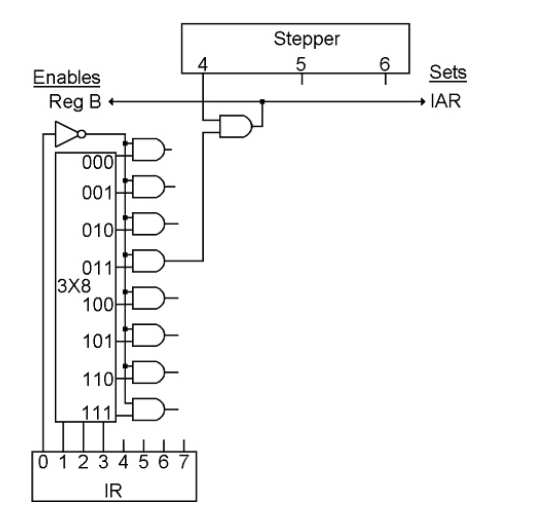
It simply moves the contents of Reg B into the IAR. Here

is the Instruction Code for it:



**The computer is executing a series of instructions in RAM, one after the other, and suddenly one of those instructions changes the contents of the IAR. What will happen then?** The next instruction that will be fetched will not be the one that follows the last one. **It will be the one that is at whatever RAM address was loaded into the IAR.** And it will carry on from that point with the next one, etc. until it executes another jump instruction.

The wiring for the Jump Register instruction only needs one step. In step 4, the selected register is enabled onto the bus, and set into the IAR, and that is all. If we wanted to speed up our CPU, we could use step 5 to reset the stepper. But to keep our diagram simple, we won’t bother with that. Steps 5 an 6 will do nothing.



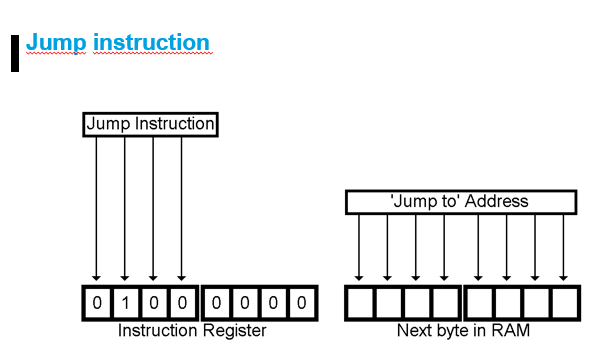
**Another Way to Jump**

Uitleg en implementatie van de onvoorwaardelijke sprong naar een adres dat in het programma is opgenomen.

This is another type of Jump instruction. It is similar to the Data instruction in that it uses two bytes. It replaces the IAR with the byte that is in RAM

immediately following the instruction byte, thus changing where in RAM the

next and subsequent instructions will come from.



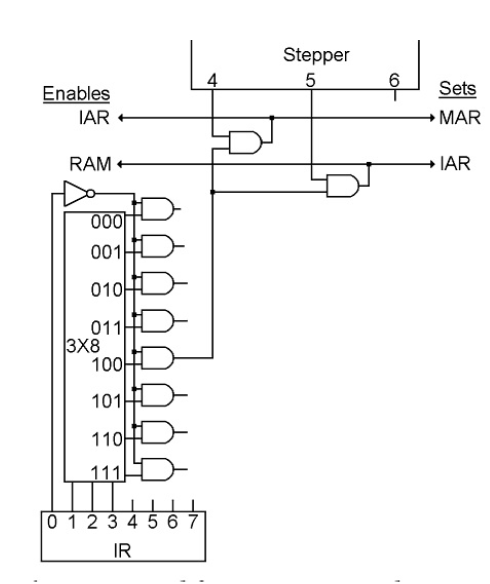
This exact type of Jump instruction is just called a "Jump." It is useful when you know the address that you are going to want to jump to, when you are writing the program. The Jump Register Instruction is more useful when the address you are going to want to jump to is calculated as the program in running, and may not always be the same.

One of the things you can do with a Jump instruction is to create a loop of

instructions that execute over and over again. You can have a series of fifty

instructions in RAM, and the last instruction "Jumps" back to the first one.

Like the Data instruction, the IAR already points to the byte we need. Unlike the Data Instruction, we don't need to add 1 to the IAR a second time because we are going to replace it anyway. So we only need two steps. In step 4, we send IAR to MAR. In step 5 we move the selected RAM byte to the IAR. Step 6 will do nothing.



**The Third Great Invention**

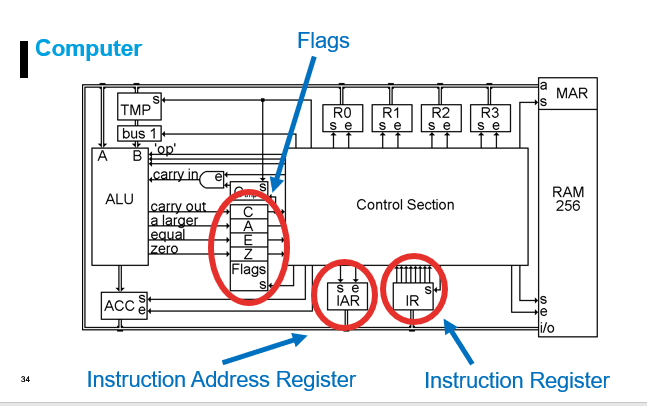
Uitleg en implementatie van het concept van voorwaardelijke sprongen gebaseerd op 'flag' bits in het statusregister.

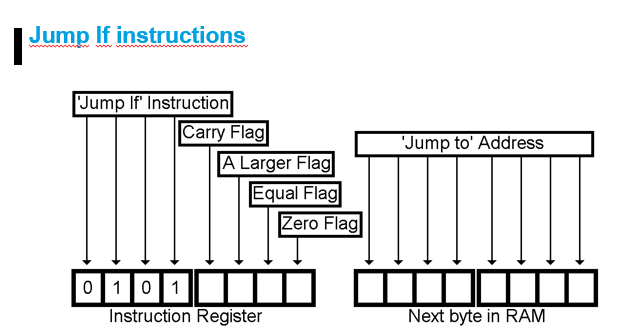
This is just like the Jump Instruction, but sometimes it jumps, and sometimes it doesn't. Of course, to jump or not to jump is just two possibilities, so it only takes one bit to determine which will happen. Mostly what we are going to introduce in this chapter is where that one bit comes from.

Do you remember the 'Carry' bit that comes out of, and goes back into the ALU? This bit comes either from the adder, or from one of the shifters. If you add two numbers that result in an amount that is greater than 255, the carry bit will come on. If you left shift a byte that has the left bit on, or right shift a byte that has the right bit on, these situations will also turn on the ALU's carry out bit.

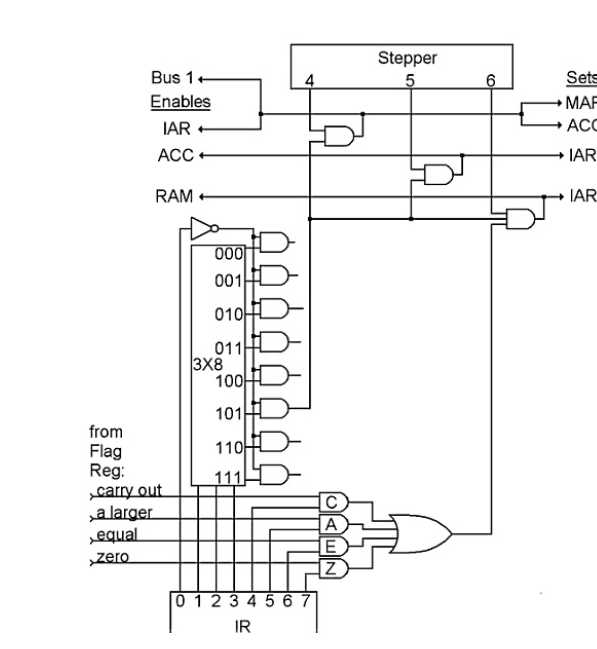
There is also a bit that tells us if the two inputs to the ALU are equal, another one that tells us if the A input is larger, and one more bit that tells us if the output of the ALU is all zeros.

These bits are the only things that we have not yet found a home for in the CPU. These four bits will be called the "Flag" bits, and they will be used to make the decision for a "Jump If" instruction as to whether it will execute the next instruction in RAM or jump to some other address.

The "Jump If will jump or not depending on something thathappened during the ALU instruction. 



Meer inlezen indien gewenst.



Let op bij die or gate. Er is gebruik van een or gemaakt, omdat we niet zeker weten of al die CAEZ waar zijn. Hierdoor kan natuurlijk de and gate bij de IAR niet geset worden. Dus daarom de or.

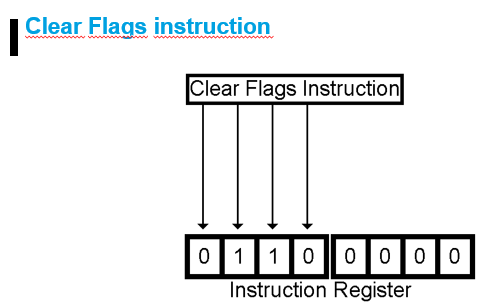
**The Clear Flags Instruction**

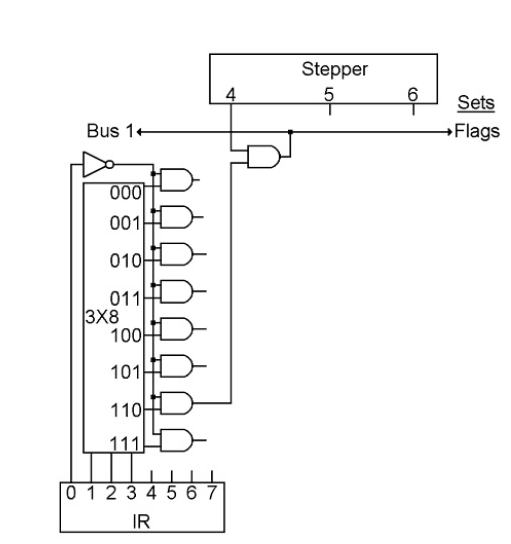
Uitleg van de opdracht waarmee 'flags' in het statusregister op 0 gezet kunnen worden.

When you do addition or shifting, you have the possibility of getting the carry flag turned on by the operation. This is necessary, we use it for the Jump If instruction as in the previous chapter.

The Carry Flag is also used as an input to the addition and shift operations. The purpose of this is so you can add numbers larger than 255 and shift bits from one register to another.

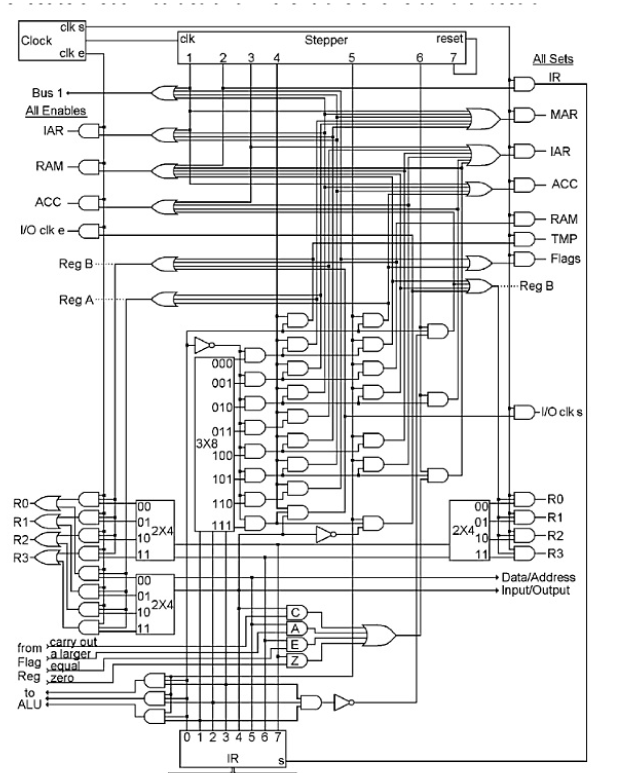
The problem that arises is that if you are just adding two single-byte numbers, you don't care about any previous Carry, but the Carry Flag may still be set from a previous operation. In that case, you might add 2+2 and get 5!

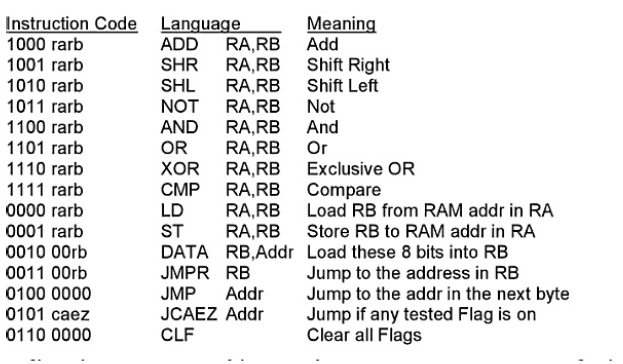




**Ta Daa!**

Overzicht van de complete aansturingseenheid van de CPU, met een tabel van alle tot dusverre behandelde opdrachtcodes.





Dit is blz 122  
Moet je deze opdracht codes kennen?