Embedded Systems Notes:

Embedded Notes:

Development Environment:

Editor -> Assembler / Compiler -> Linker -> Debugger / Emulator ->)

In-circuit emulator:

Emulator sometimes is different than actual hardware

Flash Programmer:

Standard C + Extention:

Const – something that doesn’t change

Volatile – a variable that can change

P1IN: 8 bit

P1IN = 0b11110000;

(0b means it’s a binary number)

Port 1 Input =

BIT4 = 0b00010000;

Bitwise: Logical:

& &&

| ||

^

P1IN && BIT4 = true

P1IN & BIT4 = BIT4

11110000

00010000

---------------

00010000

We use a mask operation to check if a bit is 1, or to modify it.

P1IN & BITx

P1OUT = P1OUT | BIT4;

P1OUT &= ~BIT4

And P1Out with NOT bit4

BIT4 = 00010000

~Bit4 = 11101111

P1OUT |= BITX; // SET BIT X

P1OUT &= ~BITX; // CLEAR BIT X

P1OUT ^= BITX; // INVERT BIT X

P1OUT |= (BIT5 | BIT4); // SETS ONLY BIT 4 AND 5

P1OUT |= ~(BIT5 | BIT4); // CLEARS ONLY BIT 4 AND 5

\_\_no\_init volatile union {

unsigned short TACTL;

struct{

unsigned short TALFGL 1;

unsigned short TASSFL: 2;

}

TACTL\_bit;

}@ 0x0160;

set TACTL\_bit.TALFG = 1;

clear …. = 0

Toggled ….. ^= 1;

Unsigned short name = 8 bit variable from value 0 to 255

Signed short name2 = 8 bit from value -127 to 128

Char. -8 bit

Volatile unsigned char P1IN;

@ 0x0030

Intrinsic functions:

Status register (SR)

‘Global Interrupt Enable’

\_\_enable\_interrupt();

\_\_disable\_interrupt();

Lecture 2:

Volatile – side effects when you access a variable.

Behavior:

Undefined behavior – crashes program

Implementation-defined behavior - program changes based on computer

Unspecified behavior – when the C language gives you more than one way to do something, so you “pick one and go with it”.

Locale-specific behavior – program changes based on location

Const

Volatile

Restrict

Don’t do this, your program will crash:

Const char\* in = “Hello world”l

Char \*p = (char \*)Exm[3];

\*p = ‘L’;

Implementation-Defined – changes based on the computer and compiler, etc:

volatile int a;

a = 10;

C language allows compiler to add padding between members of a struct, so the size won’t be accurate.

Undefined behavior:

struct my\_float{

int sign : 1; // one bit wide

int exponent: 8;

int mantissa: 23;

}

Don’t ever use bit fields unless you want your thing to break in unexpected ways.

Sometimes companies increase sizes (int = 4 bytes)

Short – 2 bytes

Int – 2 bytes

Long – 4 bytes

Long long – 8 bytes

Fixed width integers – 7.18

Stdint.h

“minimum width integer that satisfies that condition”

int\_leastN\_t uint\_leastN\_t

N = {8,16,32,64}

Int\_fastN\_t uint\_fastN\_t

In program:

#include <stdint.h>

Int\_least16\_t counter;

Int b;

Lecture 3

Memory Map

|  |  |
| --- | --- |
| 0x200 | RAM |
| 0x1ff  0x100 | 16-bit Peripheral modules |
| 0x0ff  0x010 | 8-bit Peripheral modules |
| 0x00f  0x000 | SPR – Special purpose register |

Up ^

Down v

|  |  |
| --- | --- |
| 0xffff | Interrupt vector Table |
| 0xffdf | ROM / Flash |

CPU Core:

1. Instructions:

.b – bytes (only 8 bits)

.w – words, default (16 bits)

.a – extended address mode, MSP430X

2. 16 Registers – each 16-bits wide (20 bits wide for MSP430X)

3. 16-bit ALU – (20 bit ALU for MSP430X)

MSP430 Register Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Register | Primary | Secondary | Description |  |  |
| r0 | PC |  | Program Counter |  |  |
| r1 | SP |  | Stack Pointer |  |  |
| r2 | SR - |  | Status Register |  |  |
| r3 | CG2 |  | Second Status Register |  |  |
| r4~r15 |  |  | General Purpose Register |  |  |
|  |  |  |  |  |  |

A heap only stores variables when you’re running a process. For example, we have a for loop where a variable I is dynamically created, but you don’t need it when you finish. A stack stores all of your instructions for the entire program.

“**Stack** is used for static memory allocation and **Heap** for dynamic memory allocation, both stored in the computer's RAM . Variables allocated on the **stack** are stored directly to the memory and access to this memory is very fast, and it's allocation is dealt with when the program is compiled.”

“This is used when the amount(size) of **memory** is variable and is known only during run-time. **Dynamic** allocation is achieved using certain functions like malloc(), calloc(), realloc(), free in C and "new", "delete" in C++. **Static Memory** Allocation - **memory** allocated at compile time in stack or other data segments.”

Application binary interface – ABI – we pass parameters to a function and tells the library where to look for the arguments, exactly the way we pass an address.

MIPS:

Globe $s0-$s7

Temporaries $t0-$t9

Argument $a0-$a3

Return values $v0, $v1

Find a file defined by TI called slaa534 that defines each register.

In assembly, semicolon represents a comment.

Addressing Mode:

http://harijohnkuriakose.blogspot.com/2010/10/addressing-modes-in-msp430-family.html

- Register Mode

mov.w r4, r5 ; move r4 into r5

- Indexed Mode

mov.w 2(r4), 6(r6) ; copy second index of r4 into 6th index of r6

; r6[3] = r4[1]

location in memory pointed by r4 + 2 gets copied to the location pointed to by r6+6

- Symbolic mode

mov.w lb1, lb2

lb1: /\*…\*/

lb2:/\*…\*/

Register Mode

mov.w R4,R5 ; move (copy) word from R4 to R6

It is the fastest, with only 1 machine cycle needed.

Any of the 16 registers can be used as source or destination.

Special cases:

• PC - it will be autoincremented before it is used as source

• Both PC and SP must be even, because they are always used as words. so LSB discarded if they are used as destination

• CG2 - it reads 0 as source

for byte operations:

• operand is taken from lower byte only

• writing is performed to lower byte only, upper byte is cleared

To use the upper byte in a regiser as source, 'swpb' may be used.

Indexed Mode

Similar to arrays.

mov.b 3(R5),R6 ; load byte from address 3+(R5) into R6

Here, base address is 3.

Indexing can be used for the source or destination part.

Symbolic Mode (PC Relative)

When PC is used as the base address in the indexed mode, its called symbolic mode by TI. The offset to be added to the PC is given as the constant.

mov.w Loop,R6 ; load word Loop into R6

Assembler replaces this as:

mov.w X(PC),R6 ;

where X = Loop - PC, is the offset in this case. It is caluclated by the assembler, which also performs autoincrementing of PC.

In MSP430, absolute addressing can reach all the memory map. The symbolic mode is mainly meant for MSP430X, etc.

Absolute Mode

This is a special case where the constant in the indexed mode is the absolute address of the data. Since the constant is already the final address, the base must be taken as an address of 0. Usually the SR is selected for this purpose. It behaves as 0 when used as the base, i.e, this is one instance when the SR behaves as a constant generator (CG1).

Absolute addressing is shown by the prefix &.

mov.b P1IN,R6 ; load byte P1IN into R6

It is replaced by the assembler as:

mov.b P1IN(SR),R6 ;

P1IN is the offset, and SR behaves as 0.

SP-Relative

This is not a separate mode in itself. At any time, any value pushed into the stack previously can be accessed, by offseting a suitable amount from the SP. For example:

mov.w 2(SP),R6 ;

Indirect Register Mode

This is available only for the source. It is indicated by the sign @. It means that the contents of a register is used as the address of the operand, i.e, the register contains a "pointer" to the actual operand.

mov.w @R5,R6 ; load word from address pointed to by R5

This is similar to indexed addressing with base address 0. It saves a word of program memory, hence makes it faster.

This mode cannot be used for destination. Using indexed addressing instead:

mov.w R6,0(R5) ; store word from R6 into address 0+(R5)

There is a penalty that a word 0 must be stored in the program memory, and fetched. The constant generator cannot be used.

Indirect Autoincrement Register Mode

This is also available only for the source. It is indicated by a @ in the front, and a + as suffix. Here, the register is used as a pointer as in the indirect register mode. After this, the value in the register is autoincremented by 1 if a byte has been fetched, or by 2 if a word has been fetched.

mov.w @R5+, R6

Since this mode cannot be used for destination, the indexed addressing mode must be used and then explicitly incrementing the value of the register appropriately. Obviously, two instructions would be required.

N.B.

• MSP430 only has postincrement addressing.

• In all the addressing modes, all operations on the first address are fully completed before the second address is evaluated.

Immediate mode:

mov.w #12345, r6

OAXODD will turn off LFXT1

SCG0 will turn off sDC0

SCG1 will turn of SMCLK

|  |  |  |
| --- | --- | --- |
| **Bit** | **Name** | **Description** |
| 16-19 | n/a | Reserved, mirror PC |
| 15-9 | n/a | Reserved |
| 8 | OV | Overflow bit |
| 7 | SCG1 | Status 2 bit |
| 6 | SCG0 | Status 1 bit |
| 5 | OSCOFF | Oscillator off bit |
| 4 | CPUOFF | CPU off bit |
| 3 | GIE | Global Interrupt Enable (At the beginning of the interrupt, we disable other interrupts) |
| 2 | N | Negative bit |
| 1 | Z | Zero bit |
| 0 | C | Carry bit |
|  |  |  |
|  |  |  |
|  |  |  |

Today’s Lecture:

Homework is due in 2 weeks, please turn it in on time.

What is an interrupt? The interrupt is like an alarm clock. It will notify you that something happens when you are doing something else. While it’s a simple concept, when it happens, it

makes things more complex.

Google says: “In systems programming, an **interrupt** is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An **interrupt** alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing.”

Code: Please double check that I’m typing this correctly, as I am far away from the board.

#include <msp430.h>

#ifdef \_\_GNNC\_\_

\_\_attribute\_\_((interrupt((TIMERO\_AO\_VECTOR))

#else

#pragma vector = TIMERO\_AO\_VECTOR

#endif

Void timero\_tsr(void){

P1OUT ^=0xff;

}

//WDTCTL = Watchdog Timer Control

//WDTPW = Watchdog Timer power

// TACCTL = TAC control

// GIE = Global interrupt enable

int main(void){

WDTCTL = WDTPW | WDTHOLD;

P1DIR = 0XFF;

P1OUT = 0X01;

TACCTL = CCIE;

TACCRO = 62499;

TACTL = TASSEL | MCO | ID\_3;

\_\_bis\_SR\_register(CPUOFF | GIE); // this is an i

}

Interrupt, Trap.

Exception.

An exception is very similar to an interrupt. The whole CPU deviates from its original control flow to jump to a new preset location to perform the new tasks.

1. Vectored interrupt - different interrupts have priority. (MSP430, AVR, ARM, Intel)
   1. Fixed - interrupts have a preset level of priority
   2. You define which interrupts come first and which come last. (AVRxmega, ARM -M0, ARM - M4)
2. Non vectored interrupt -

The interrupt handler picks up the interrupts. The handler first checks if the interrupt is accessible (maskable, nonmaskable, etc).

MSP430: Vectored Interrupt:c

Maskable - interrupts that can be ignored by the CPU, almost all peripheral interrupts except the watchdog.

Nonmaskable - interrupts that can’t be ignored by the CPU, if they happen, they must be taken care of. (Oscillator faults, a reset signal, etc)

System reset - PUC- power up clear , POR - Power on reset, BOR, ~RST

* 0xffeo ~ 0xffff reserved

For vector table

* This space is divided into 16-bit words.
* Then each 16-bit word contains an address of an interrupt routine.
* Position matters. Means that the higher in this table, the higher priority the interrupt is.
* The interrupt latency is 6 cycles.
* Five cycles for MSP430X
* Return from interrupt. Reti (return from interrupt)

Latency:

1. Any currently executing instruction is complete.
2. PC (Program counter) is pushed to the stack.
3. SR (status register) is pushed to the stack.
4. The highest priority interrupt is selected if you have multiple interrupts happening at the same time.
5. The interrupt requests flag resets automatically on a single source flag.
6. SR is cleared. Sr.gie = 0 sr.CPUOFF = 0
7. The contents of the interrupt vector is loaded into PC, system starts interrupt.

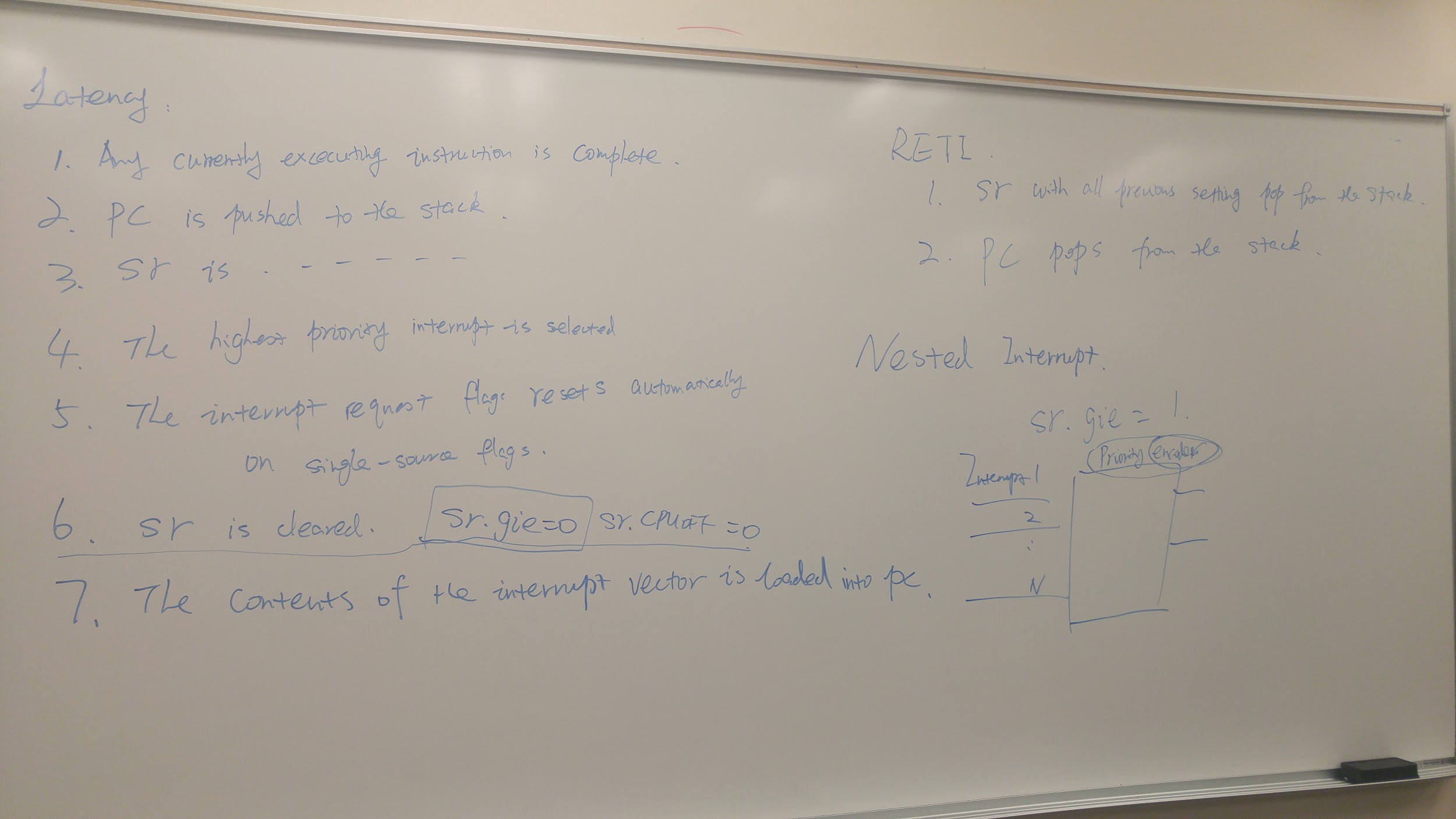
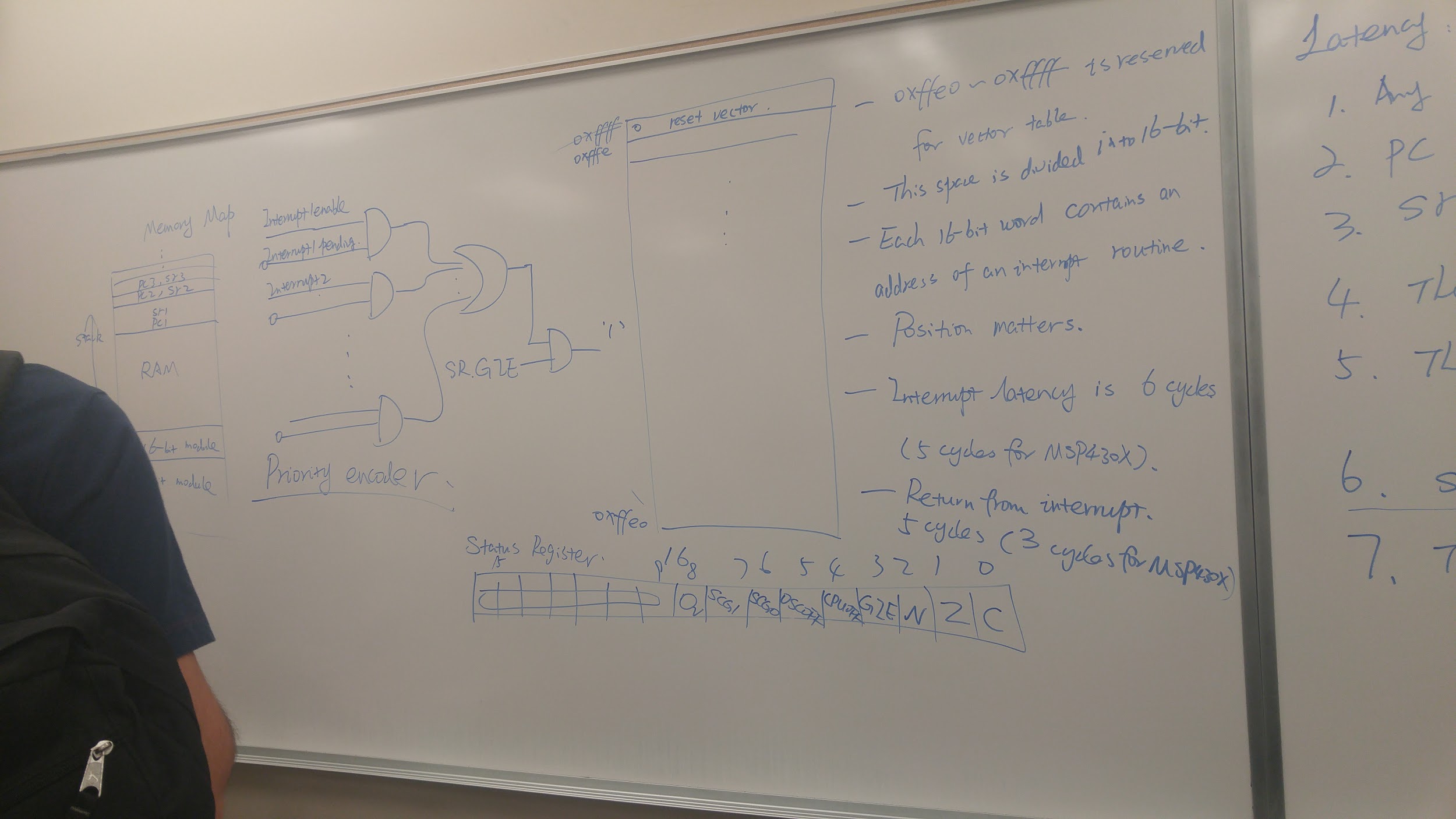
RETI:

1. SR with all previous settings is popped from the stack.
2. PC is popped from the stack.

Nested Interrupt: (my life) - during the routine, the routine may set this to 1 (sr.gie = 1). Which allows a second interrupt to interrupt the first interrupt. Too many nested interrupts can realistically crash the CPU. We want to avoid using the same type of interrupts, because they continue to pile up until there is a stack overflow.

Interrupt enable is anded with the interrupt pending.

(Int1en and Int1pend or int2en and int2pend or…) and sr.gie

**

Priority encoder - determines which interrupt to take when multiple come. Figure out how it works and how it affects the way CPu deals with interrupts. It's a part of the handler logic.

The encoder converts a bunch of signals into a value.

A **priority encoder** is a circuit or algorithm that compresses multiple binary inputs into a smaller number of outputs. The output of a **priority encoder** is the binary representation of the original number starting from zero of the most significant input bit.

|  |  |
| --- | --- |
| 0xffff  Reset vector  0xfffe  Other vectors  0xffe0 | Reset vector (highest priority) |
| 0xffdf  Flash/Rom |  |
| reti |  |
| program |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Main difference between C and assembly:

If you program in assembly language, **you need to configure everything yourself.**

Status Register:

void timer0\_isr(void){

P1OUT^=0xff;

}

Watchdog always resets the system unless you give it a password. We don’t want the watchdog to interfere with our code which is why we use the watchdog hold. In order to do that, we need to set the watchdog hold bit to 1.

WDTPW = Watchdog Password

WDTHOLD = Watchdog hold

WDTCTL = WDTPW | WDTHOLD;

0x5A00 | 0x0080

= 0x5A80

Also valid:

WDTCTL = WDTPW + WDTHOLD;

Watchdog password is predefined by TI. (5A)

Most MSP 430 chips use 5A

The watchdog hold is 80 because we want to turn on the 7th bit to hold the watchdog.

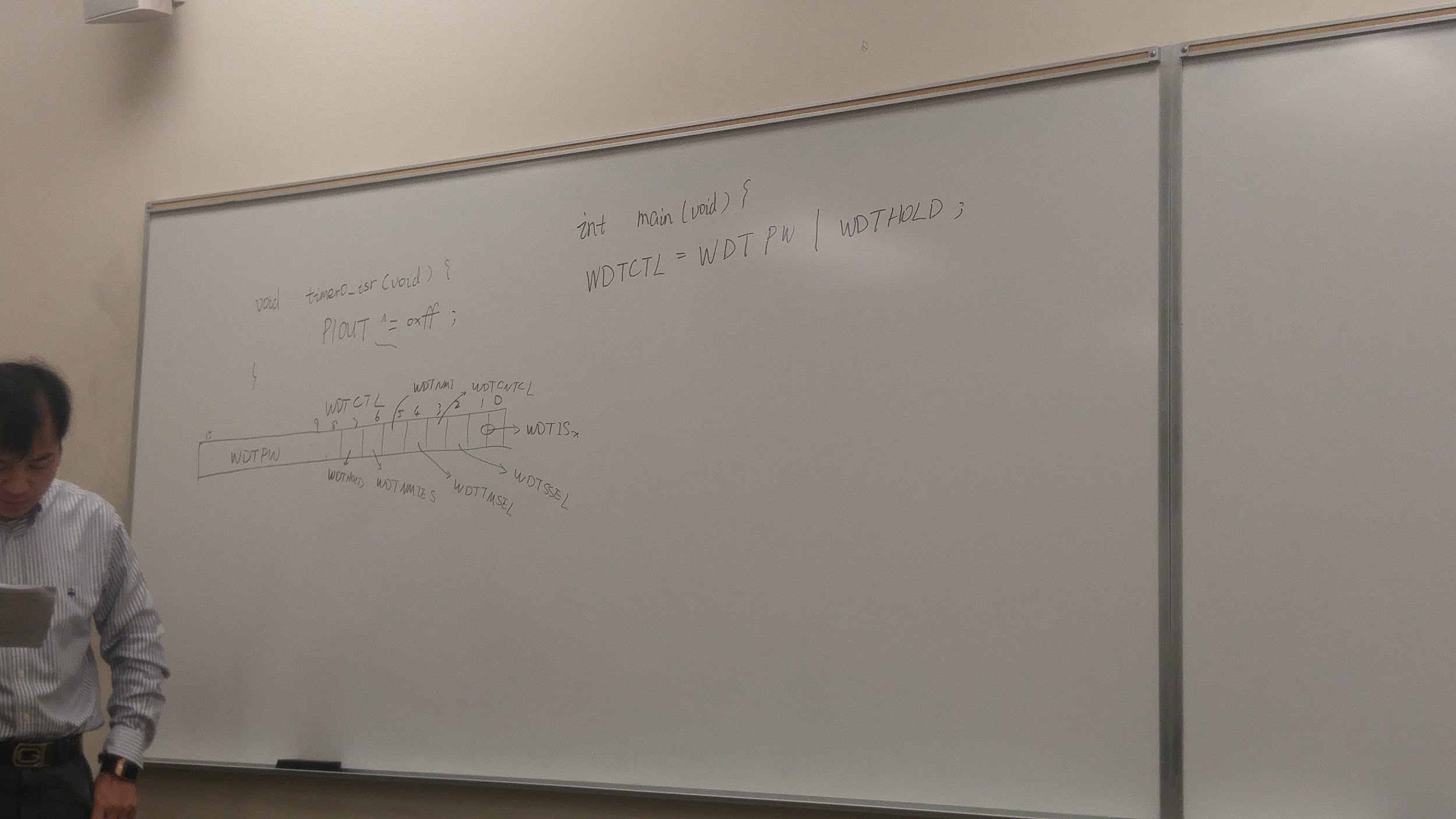
Int main(void){

// Enable and hold the WDT assembly pseudo code

WDTCTL.WDTPW = 0x5A;

WDTCTL.WDTHOLD = 0b01

}



Teacher asked us to figure out what this means:

For Timers / setting up clocks and counting to a certain value.

Timer 0 control

Timer 0 count

TACCTL0 = CCIE;

TACCR0 = 62499;

TACTL = TASSEL\_1 | MC\_0 | ID\_3;

Concise definition of these registers

Link 2: <http://www.ccs.neu.edu/home/noubir/Courses/CSU610/S07/MSP430-Clock-Timers.pdf>

Note: We don’t need to memorize the millions of registers from the data sheet for the midterm.

Multiple interrupts come, and the priority encoder decides which one to take.

Remember that an encoder takes in many bits, and reduces it to only a few. You need the priority because if the encoder had a lot of 1’s coming in, it wouldn’t know what the value of the output should be without priority.

**GPIO - General Purpose Input / Output**

1 depends on the application you’re using. It can signify 1.2 volts, 10 volts, or 100 volts.

‘1’, ‘0’, ‘X’, ‘Z’

‘Z’ - high impedance - say that a wire is dangling in the air, and you’re trying to measure the value of the impedance on one side of the wire. You get high impedance - ‘Z’. It’s basically an **open circuit.**

TTL CMOS

‘1’ input 2-5Volts 3.5-5 Volts //3.5 is the lowest input for logic 1

‘1’ output 2.7-5 volts 4.95-5 volts

‘0’ inputs 0-1.5 volts 0-1.5 volts

‘0’ output 0-0.05 volts 0~0.05volts

CMOS(5) by default

Vol = Voltage output lowest

Vil = Voltage input lowest

Voh = Voltage output highest

Vih = voltage input highest.

If output is connected to an input, your voltage comes from 4.95 to 3.5 volts, which is within an undefined region, called a noise margin. The larger the noise margin, the better.

Link 3: <http://www.ti.com/lit/ds/slas735j/slas735j.pdf>

Things you don’t learn in digsys:

Tri-state buffer -

‘Z’ high impedance

The other 3 ***magical*** digital values other than 1, 0 or X

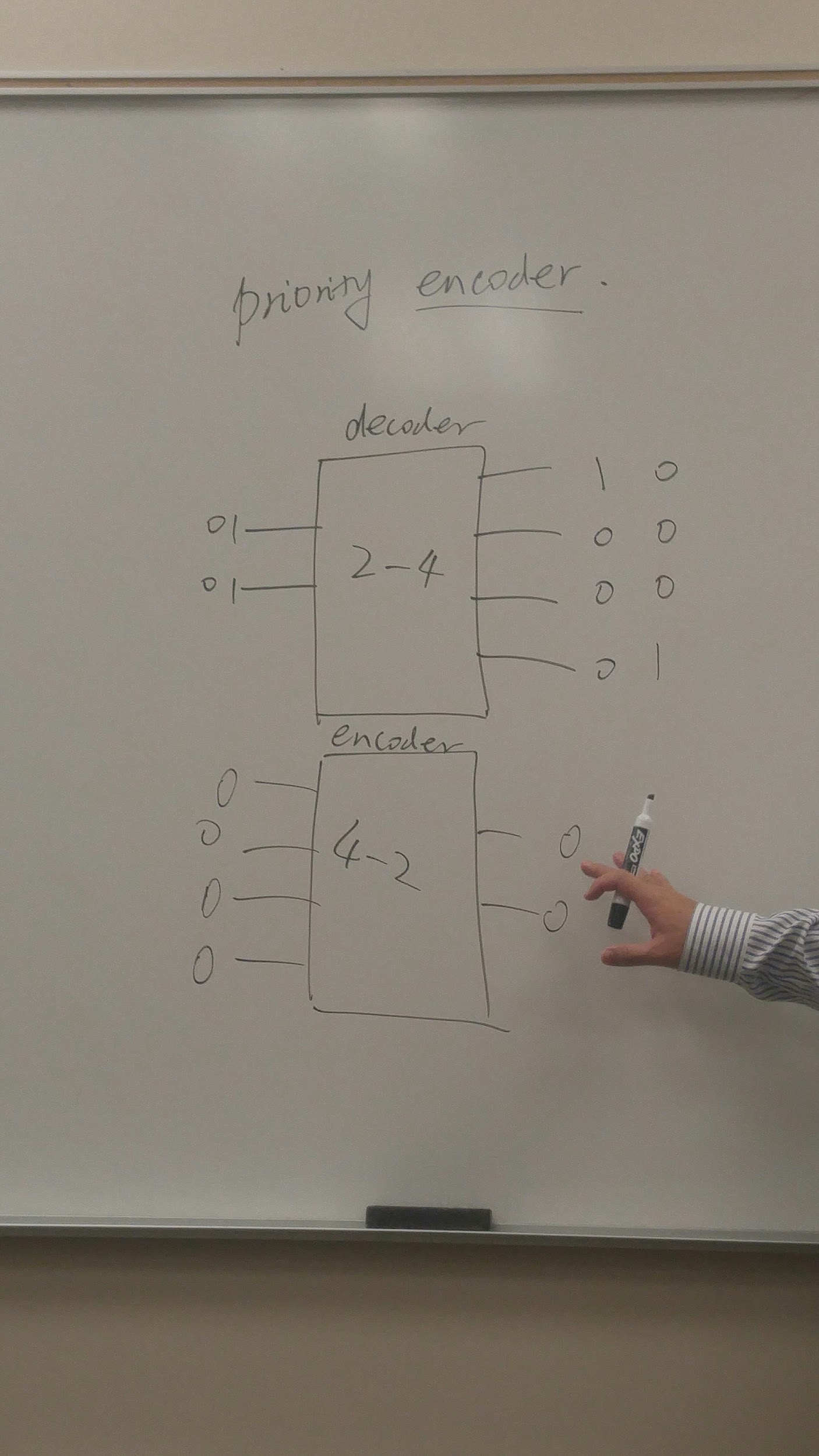
0 means connected to ground

1 means connected to Vcc

Z means it’s not connected to anything.

Think of the line that comes into the top of the buffer as a “control line”. If the value of that control line is 1, like a transistor, whatever value is in the input becomes the value of the output. If the value of the control line is 0, also like a transistor, the circuit becomes an open. The meaning of ‘Z’ means infinite impedance which is just an open.

So N mos is when the enable is normal, and P mos is when the enable has an inverter connected to it.



2/2/2017 notes:

I/O pin achieved with two buffers facing opposite directions, with one inversion on one of the buffers (input buffer). Saving and reloading the state of the pin can be achieved with flip flops

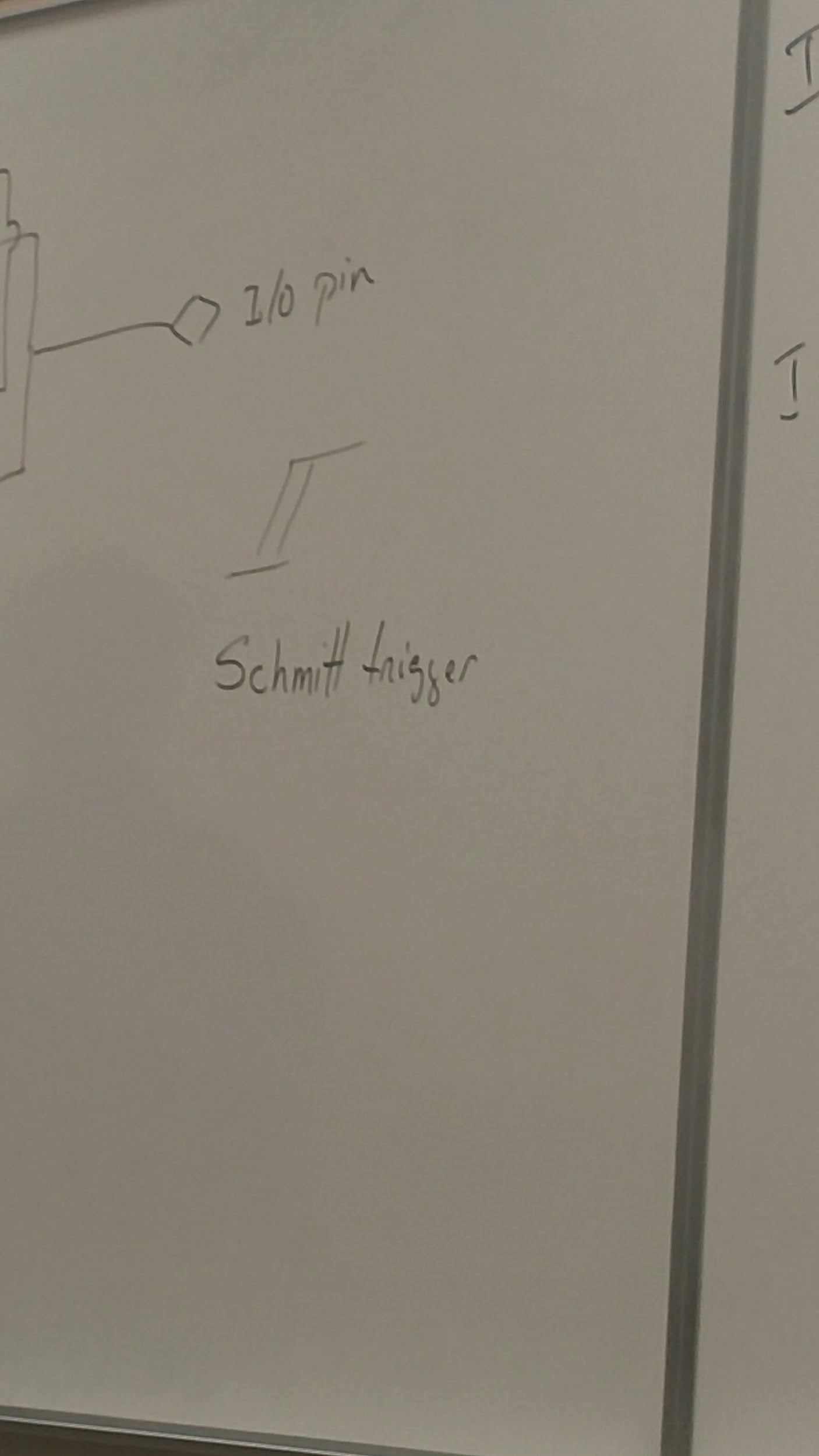
The data going out is inverted, and the data going out goes straight into the IO ko pins.

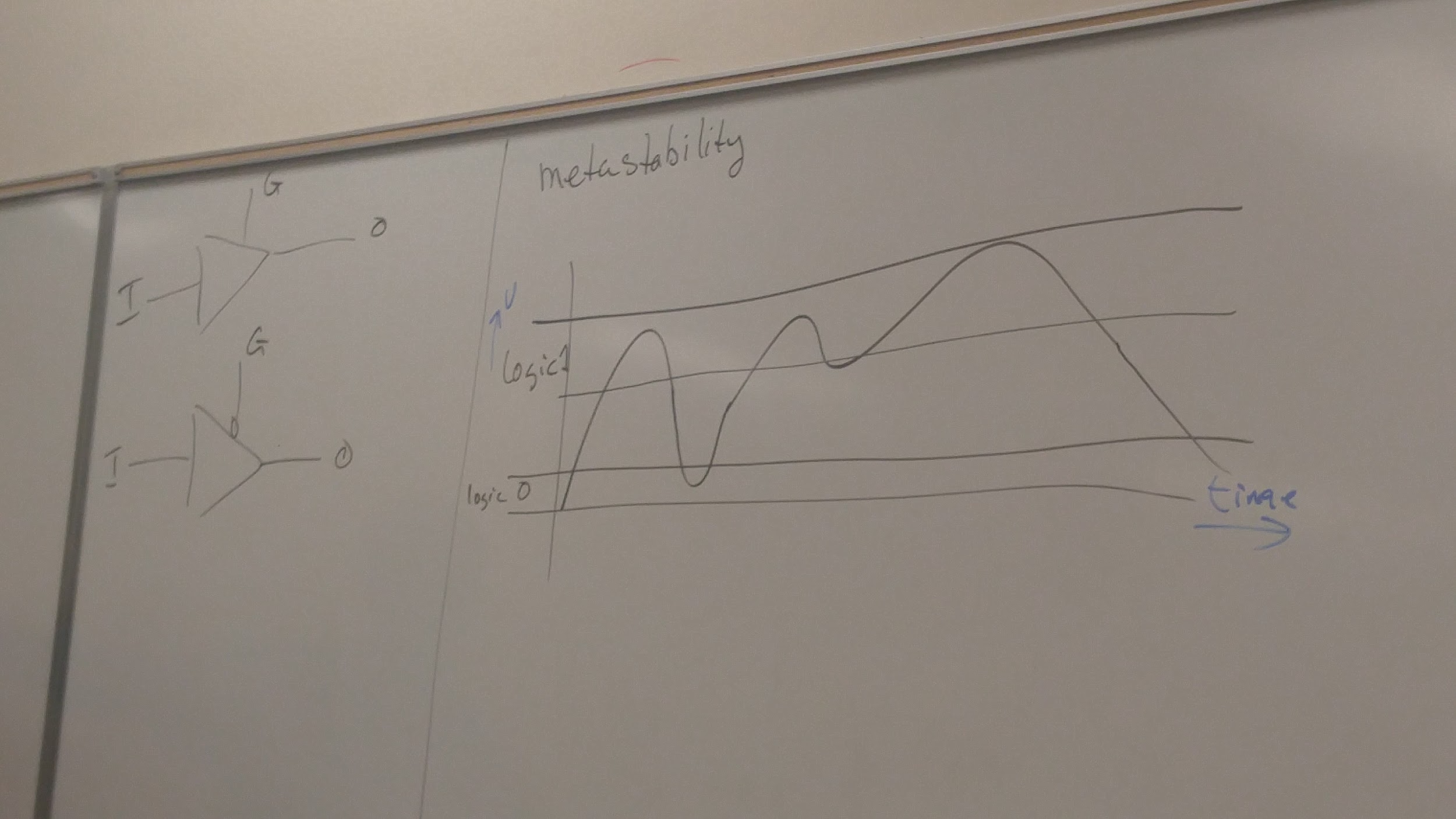
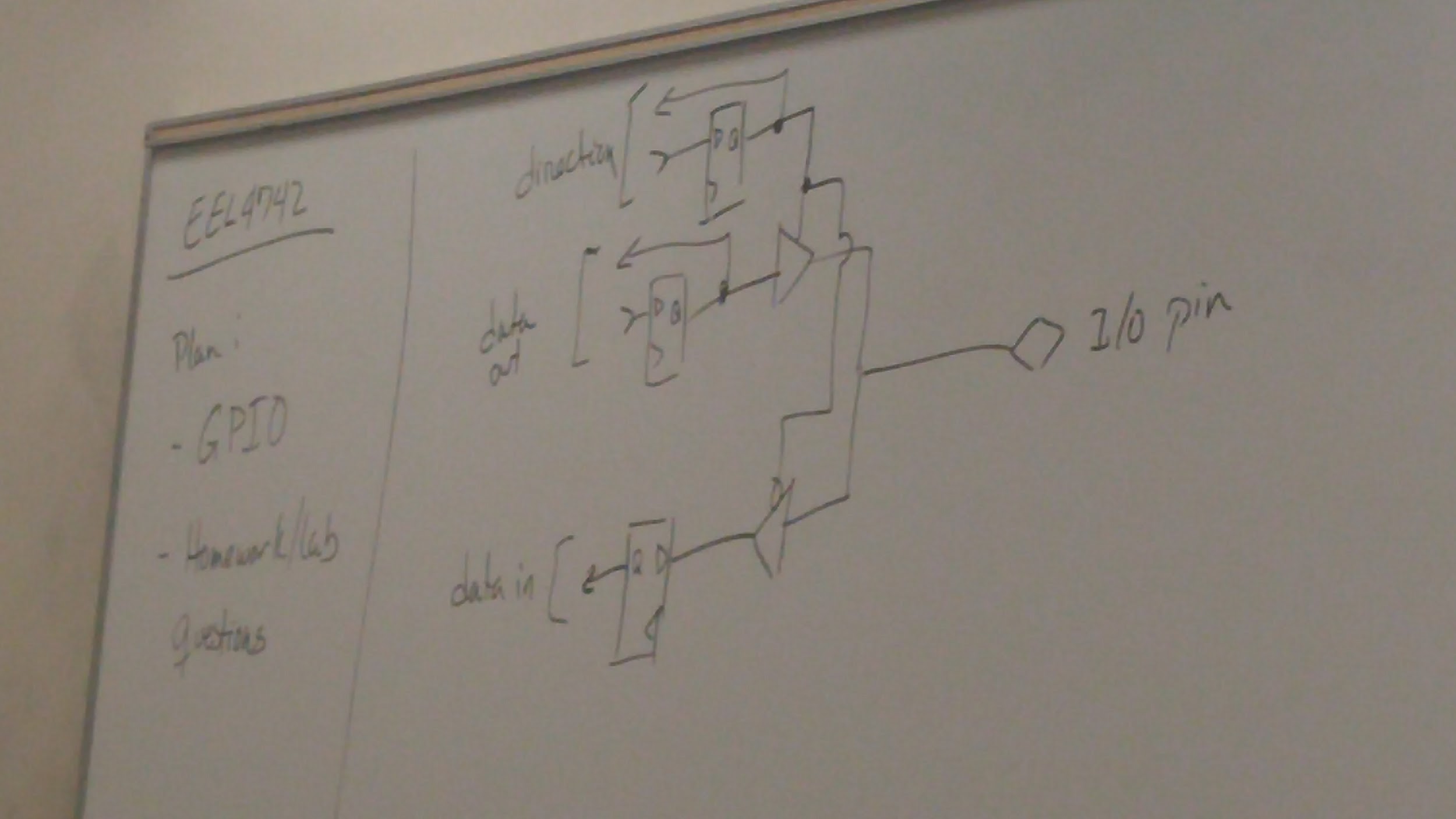
Meta-stability

Ranges of voltages are chosen to represent logic 0 and logic 1. The transition from 0 to 1 is the range between the lower and upper bound of 1 and 0 respectively, where an undefined state is. Thus instead of representing the intermediate state, TI ignores it until a complete transition between 0 and 1 occurs.

Tl;dr: If a signal remains in the indefinite state, the previously triggered signal remains active until a complete transition.

To google: Schmitt trigger





You can try predicting the voltage of a voltage level that's below your threshold, but it won't always have the effect that you want. Metastability means to take the voltages in this middle range and determine whether they should be considered a 1 or a 0. In analog communication, you can use specific algorithms to guess the voltage level, but in digital, it's easier and simpler to just wait for the next clock cycle. This noise happens so quickly that you may not always see the effect. If we had perfect vision, we'd see that the light flickers shortly before it turns on in the lab from pressing the push button.

If you can't ensure a stable logic level, you probably did something horribly wrong.

Case 1: Pull down resistor

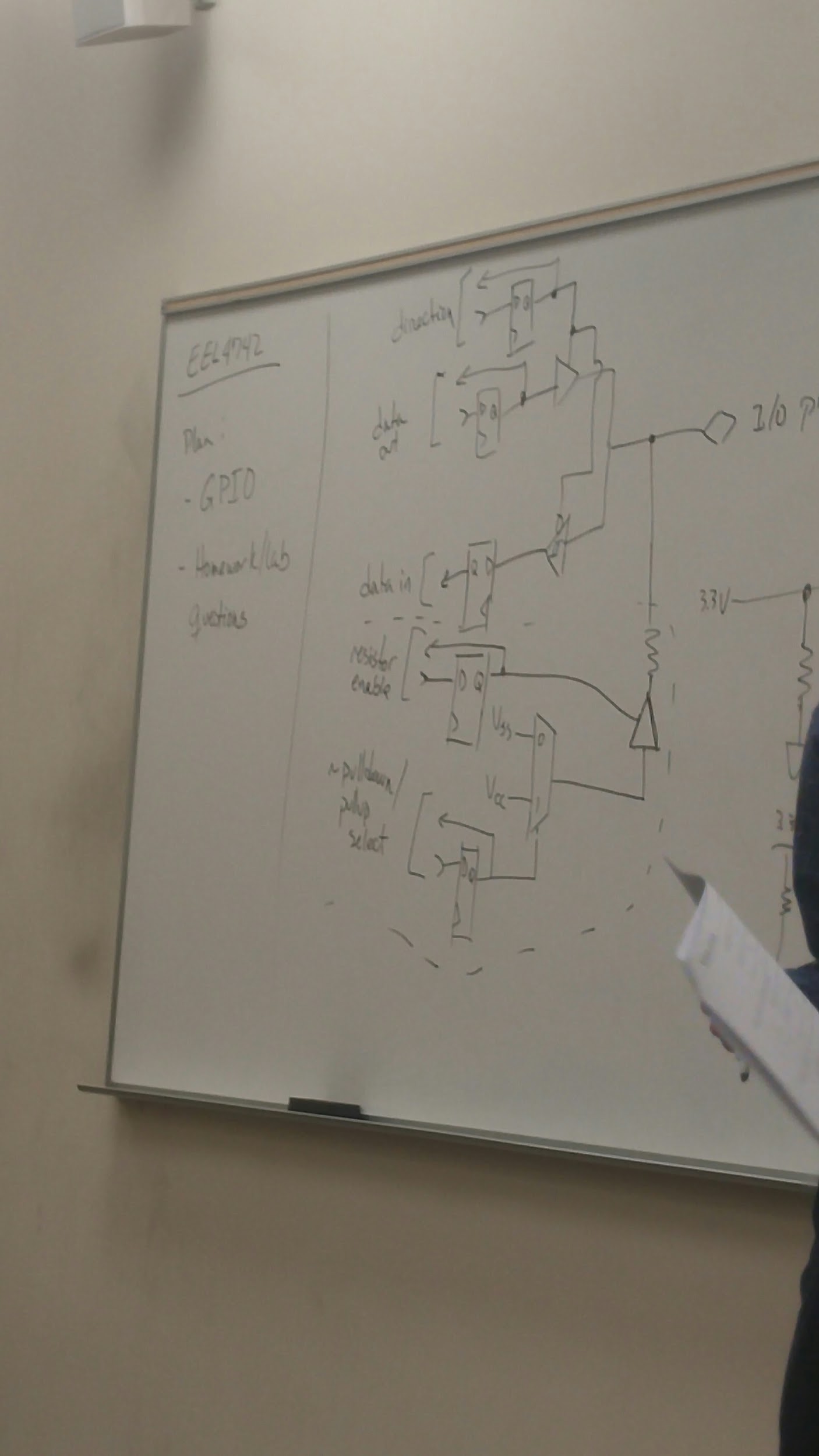
Manufacturers connect a small extra resistor to a disconnected circuit to make sure that any extra charge is a logical 0.

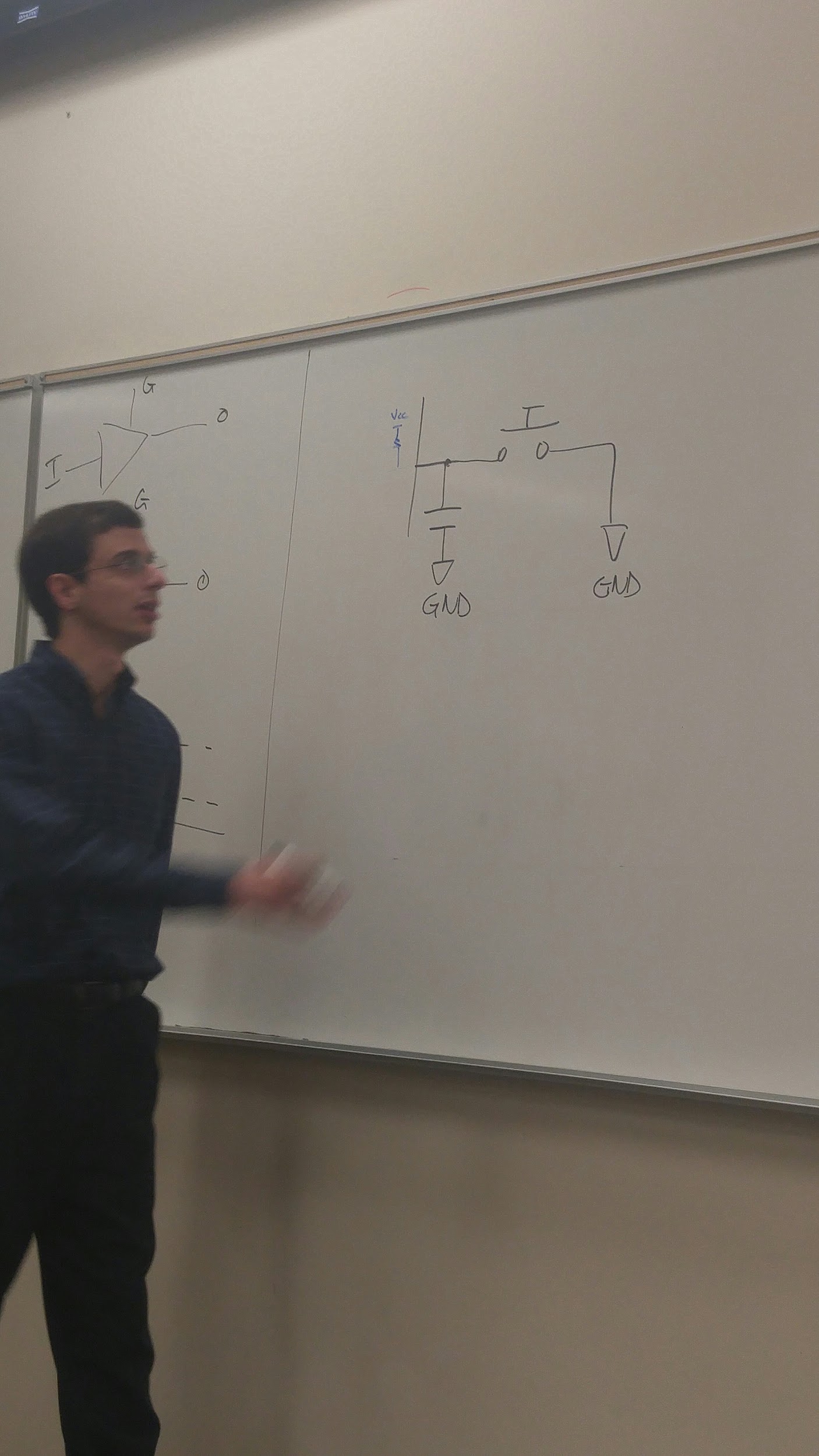
Case 2: Pull up resistor

VCC is connected to small resistor, to ensure a logical one and to prevent smoke from.Coming out of the circuit if you directly connected VCC to ground, and disconnected portion is connected to ground to ensure a logical zero.

Pull down means that the current is being pulled into the ground to equal zero.

Pull up means that the voltage is increasing in the wire.





Connecting a capacitor can reduce the noise within the logic 1 and 0, and increase metastability. However, it doesn't get rid of all of the noise, but it's more within the threshold. Schmitt trigger gets rid of the intersection of the voltage and the threshold.

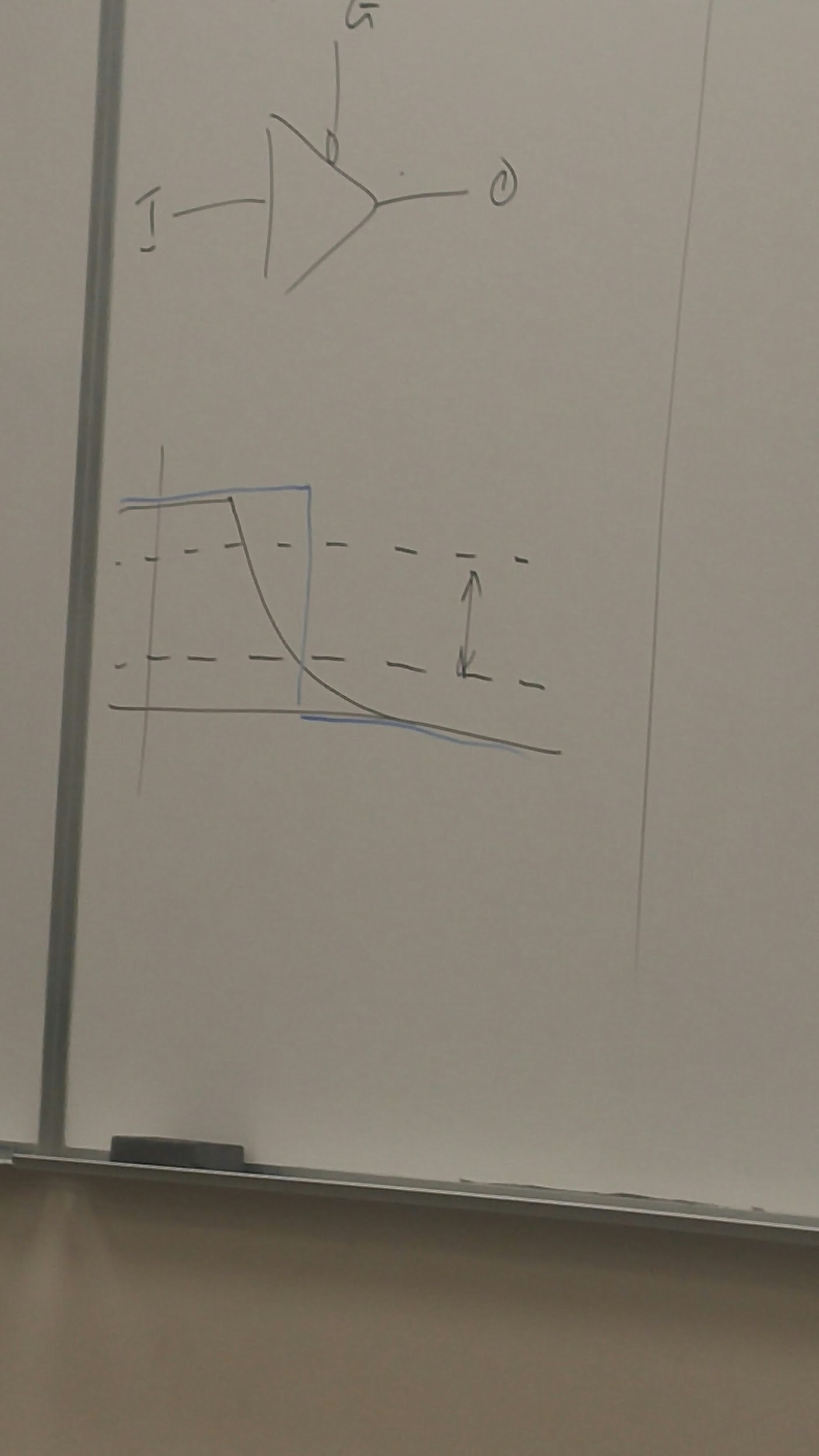
If you ever need an external Schmitt trigger, look up this number:. 74HC14 (Atmel) (question 2)

A dirty / bad programming idea:

This is lazy hardware. v

Sample 10 outputs and see if they're all 1 or 0 for the next state.

Lazy programmers should be good hw designers and vice versa



You want a capacitor that is small enough that it can discharge in time, but large enough that it gets rid of the noise.

A multiplexer or decoder is used to increase functionality within the IO pins.

ADD ALL THE FLIP FLOPS IN ORDER TO REMEMBER EVERYTHING.

(OR ADD MORE INPUTS/SELECT LINES)

**Chapter 8 datasheet**

**slau144**

Calling convention means - how the program is going to treat the registers

Libc (c Library for avr)

Register 17?

Caller - caller saves the register.

Callee - function that's called is responsible for saving the register

Test: there will be some assembly coding involved. We won't memorize instruction set. In the real world you will have the code manual next to you. It's pointless to memorize. How would you perform this operation? If this is happening, what happens if I add this extra thing?

Open notes, open book, closed neighbor.

No electronic devices.

Data sheet stuff is given in test. He doesn't believe in memorization. Great if you know it but don't memorize. 6 questions total and two extra credit.

For AVR, you can't use registers below R16 for immediate values.

Not the TA but the guy who teaches the entire class when the teacher doesn't show up:

Orlando Arias

[oarias@knights.ucf.edu](mailto:oarias@knights.ucf.edu)

Questions:

1. Does anyone understand the table above that he asked us to memorize?
2. What is a Schmitt trigger?

The Schmitt trigger keeps the previous state of where you are at until a full transition is made from 1 to 0 or from 0 to 1.

<https://en.wikipedia.org/wiki/Schmitt_trigger>

Arduino? No schmitt trigger? 74HC14.

<https://www.google.com/search?q=74hc14&espvd=2&biw=1411&bih=799&tbm=shop&source=lnms&sa=X&ved=0ahUKEwiJg9Oc1fHRAhUHKiYKHQCxC4wQ_AUICCgB&dpr=1>

Nested interrupts require a priority encoder to determine which interrupt to take first. Otherwise it could crash your system.

Test:

6 questions  
2 extra credit  
Msp430 assembly code (INTERRUPTS!!)  
Open book / notes

Homework 1 is important but no Avr on exam

No bitfields

Data addressing is on exam

No memorization - problem solving

No need to memorize memory app

Gates (sadly) won't be on exam

Tristate won't be

Instruction latency won't be

KNOW THE SYNTAX FOR INTERRUPTS. THE PROGRAM WILL BASICALLY BE ON INTERRUPTS.

No von Neumann sadly

Christian, let's make a cheat sheet for the test with lots of great syntax stuff for MSP 430 and other things that will help us.

Like #3 in the homework,

If we stored the first bits of the number into the register, and then an interrupt occurred, and then we store the next bits of the number, we would have two different numbers. This is known as a race condition.

Mutex - mutual exclusion lock - you are forbidden to access one thing while you're accessing another

Takeaway from question 3: whenever you're doing complex computations with data that could possibly change, first disable interrupts, then do the computation, then set the interrupt back how it was.

Get interrupt flag, store in temp register,

Clear interrupt flag,

Do computation,

Then set flag again

In Avr:

X uses store r27:26

Y uses load r29:28

Z uses load program memory

A timer is a fancy name of a configurable counter.

Note: If the battery doesn’t work properly on a chip, it can screw up the timer because the timer is only as constant as what powers it.

Timers are used to measure periodic events.

Counts up / down

Timers generate waveforms, and can trigger interrupts.

DRAM - if you leave it there, the leakage will eventually lose the data.

So we refresh it every 64 microseconds. The timer can determine how long has passed since the DRAM has been loaded with data.

3GHZ, need to count 64 milliseconds.

That means there are 3\*10^ 9 clocks / second.

Now, to get seconds per clock cycle, we can take 1/ (3\*10^9) = 0.33333 nanoseconds / clock cycle.

We need to get to 64 milliseconds.

So we can divide 64 milliseconds / 0.3333 = 1.92\*10^8 seconds that we need to count to.

Now figure out how much space is needed to store the above number.

2^n = 1.92\*10^8

N is about 28 bits.

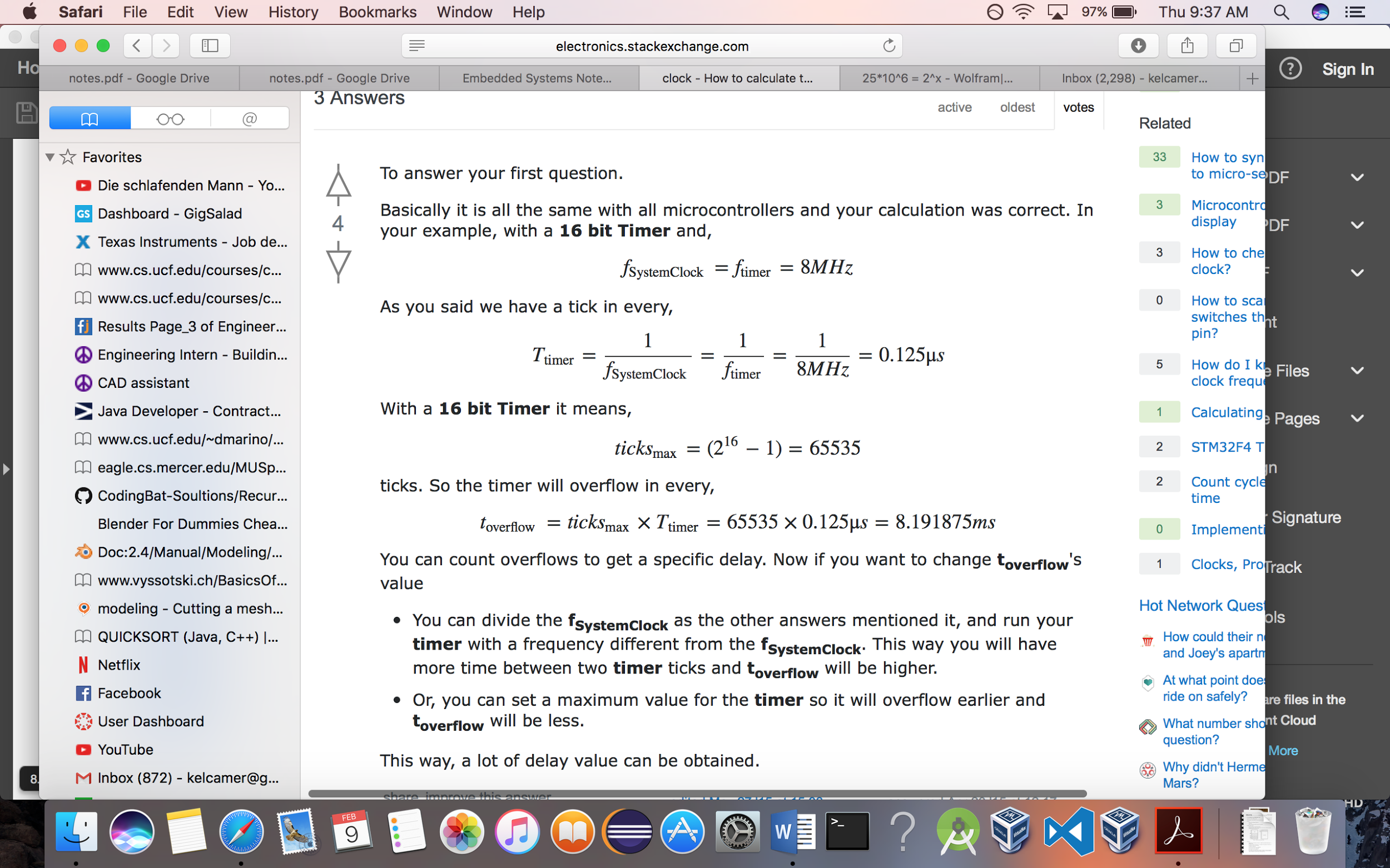
25MHz = 25 \* 10^6

1/ (25\*10^6) = 4 \* 10^-8

1 second / (4\*10^8) seconds = 25\*10^6

2^n = 25\*10^6

N = 25 bits (coincidentally)



x86:

MOST IMPORTANT:

Each channel is a timer.

Channel 0 is connected to Programmable Interrupt Controller (PIC). This is used for using interrupts. Uses NEC555 Timer - uses an external capacitor - resistor network to generate pulses. (RLC Circuit). Like a sin wave.

Channel 1 is used to refresh DRAM

Channel 2 is connected to the PC speaker / internal buzzer.

AVR:

Timer 0 is an 8 bit timer. (they ran out of space)

Timer 1 is a 16 bit timer.

MSP430:

Timer\_A, Timer\_B

Both timers are 16 bits.

Building a Timer:

1. Sequential Circuit - can store values  
   Combinational Circuit - can not store values.
2. T Flip flop - Toggles the value based on rising clock

T flip flops. Basically each new falling edge toggles the next bit. 000, 001, 010, 011, 100, 101, 110, 111. This is called an Asynchronous clock because we don’t use the same clock signal for each T flip flop. Propagation delays cause problems with asynchronous clocks. Thus, this is unfortunately not realistic for a larger design, so we need another solution. This is also called a Ripple counter.

How Timers actually work - the simple way that I figured out:

TACCTL means Timer A Capture Control.

CCIE means Capture Control Interrupt Enabled.

This first line of code basically turns the timer interrupt on.

TA0CCTL0 = CCIE;

Then you get to TA0CTL = TASSEL\_2 | MC\_1 | ID\_3;

TASSEL means Timer A Source Select. (I am not sure what the EL means).

Basically within TASSEL, there are 4 modes:

00 TACLK – Timer A Clock

01 ACLK – a low frequency 32kHZ frequency that is used to simulate real time. (For me, it took 5 seconds to blink the light).

10 SMCLK – Subsystem clock allows you to control the frequency that the timer operates and also allows you to control which value you want to count to.

11 INCLK – In clock.

MC means mode control which determines whether the timer stops, counts up, or counts up then down.

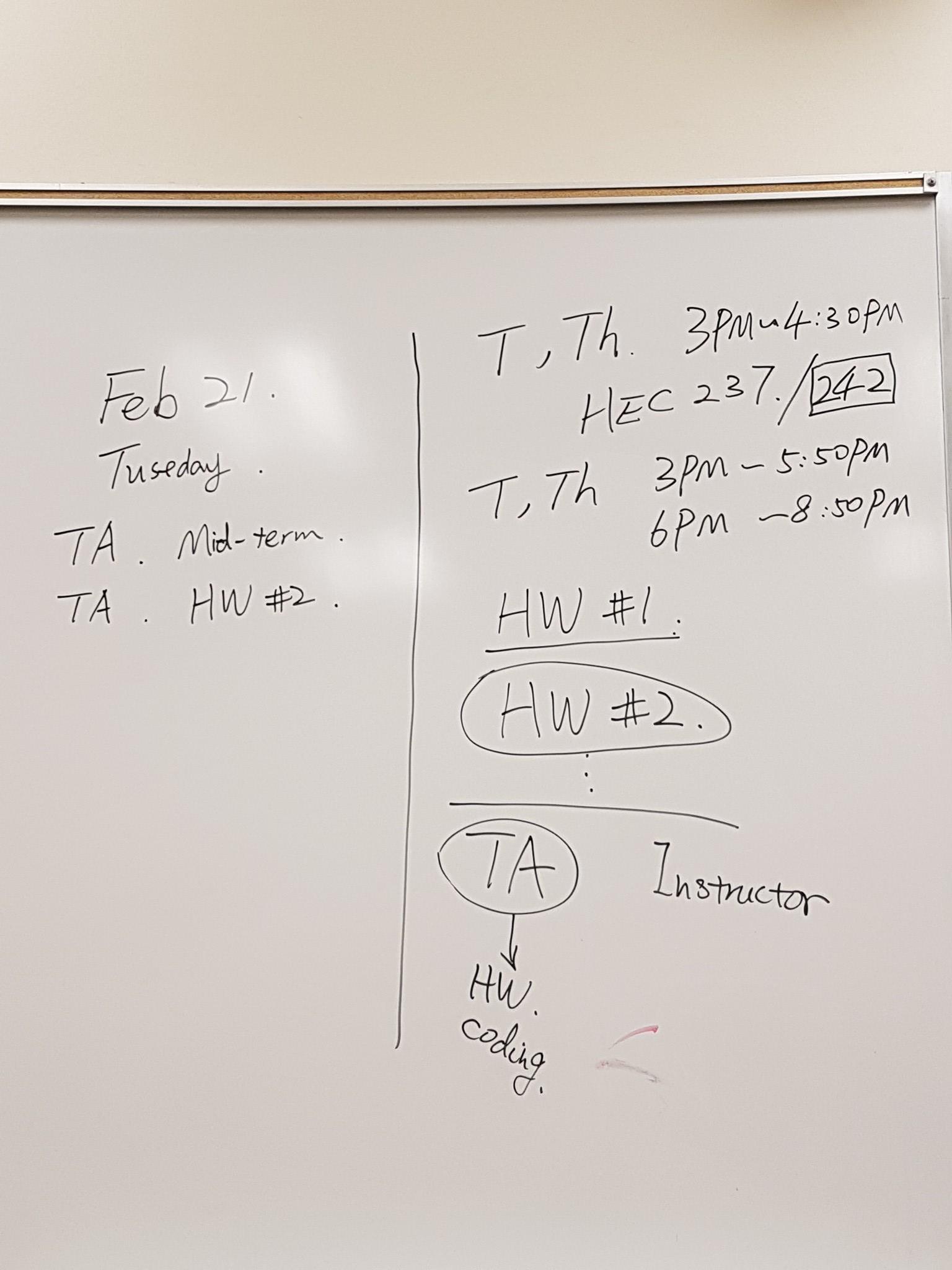
ID means input divider which changes the divisor of the input frequency.

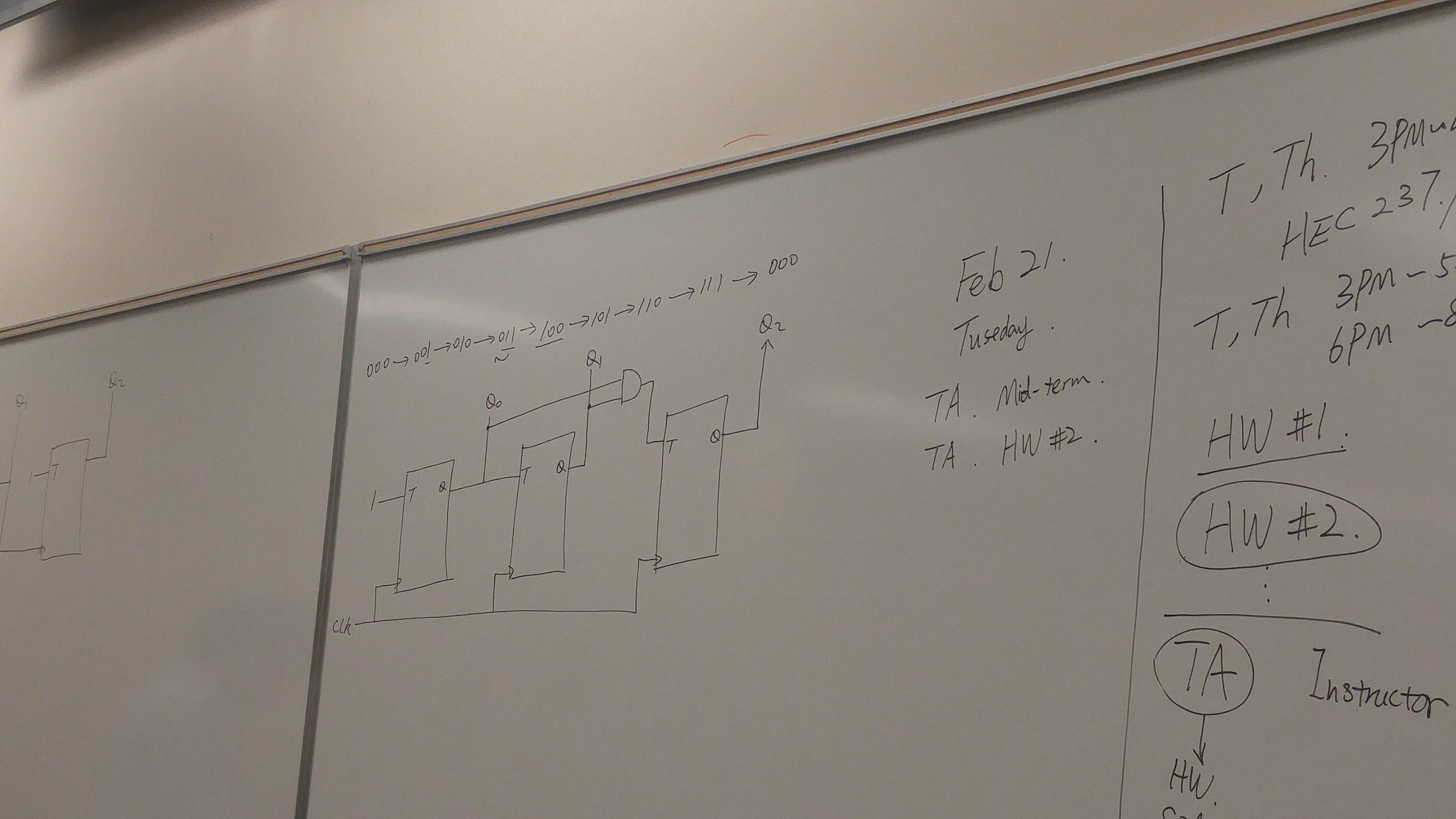
If you’re using TASSEL\_2 (source select 2), then the timer counts up to whatever value is stored in the variable called: TA0CCR0

Go to this link for more info:

[Timer Notes that Actually Make Sense](http://coecsl.ece.illinois.edu/me461/Lectures/ME461_L06_Timers.pdf)

TA office hours:





First but toggles every clock cycle. Second but toggles if the first bit is a 1, and when the clock is a 1. Same for third bit.

There's a small propagation delay.

This is a synchronous design.

000 the counter starts here

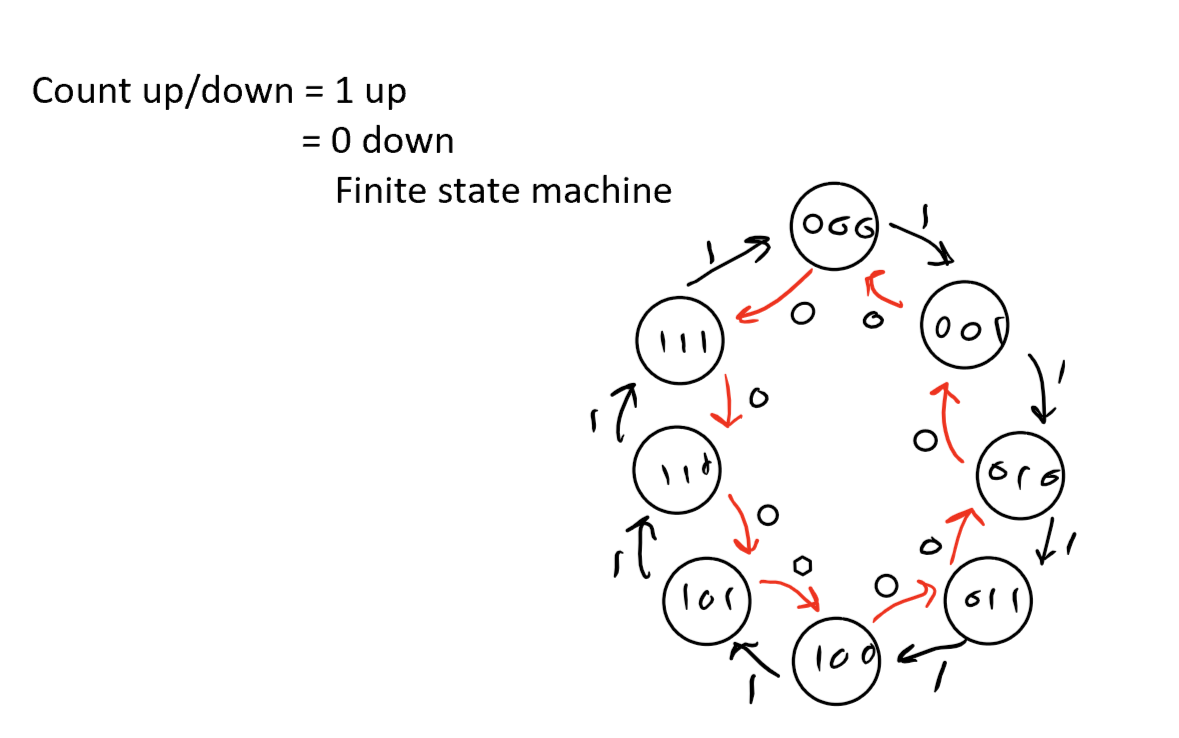
001. Last bit toggles

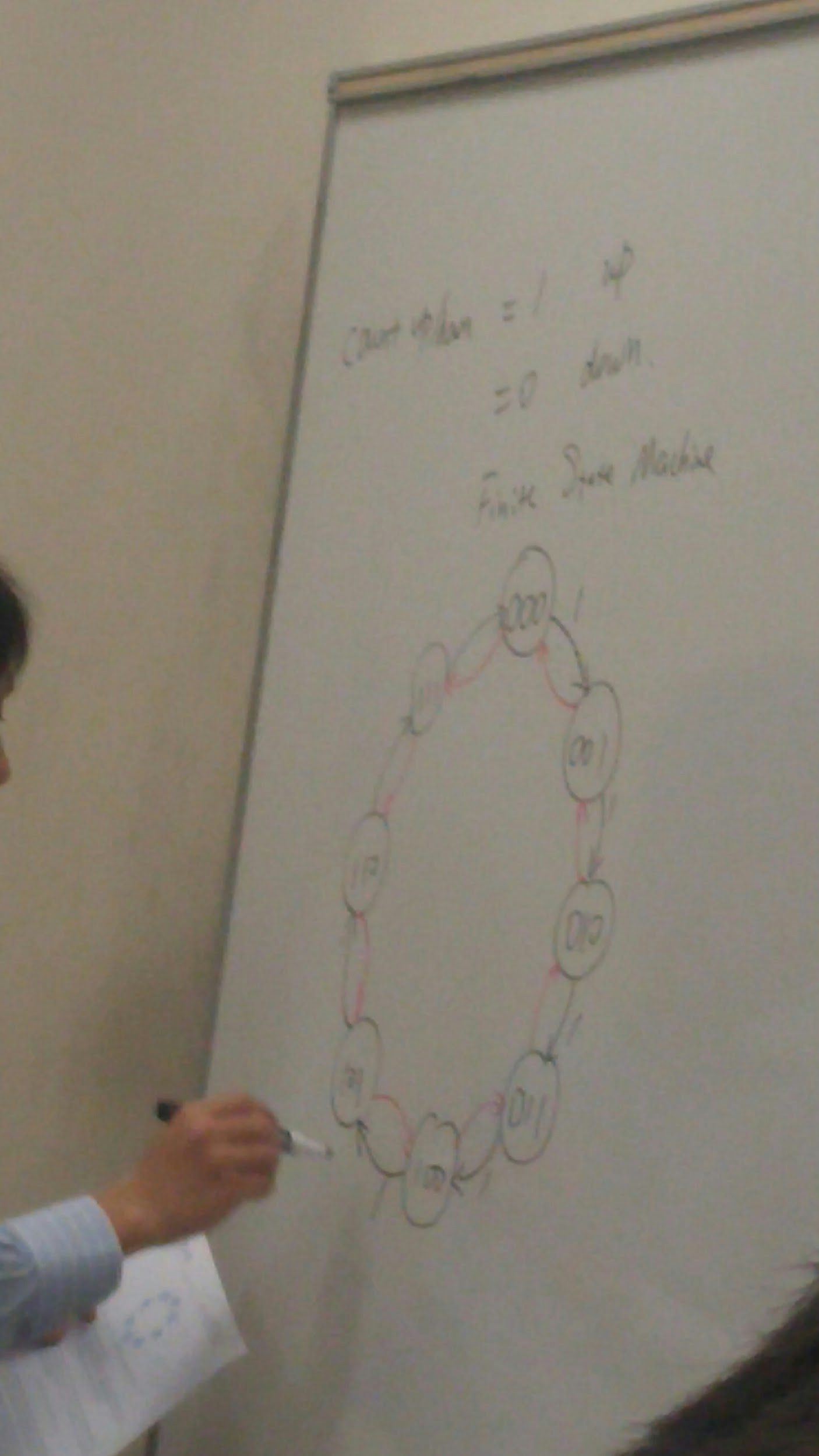
010. Last bit toggles back, but the previous 1 turns on next bit.

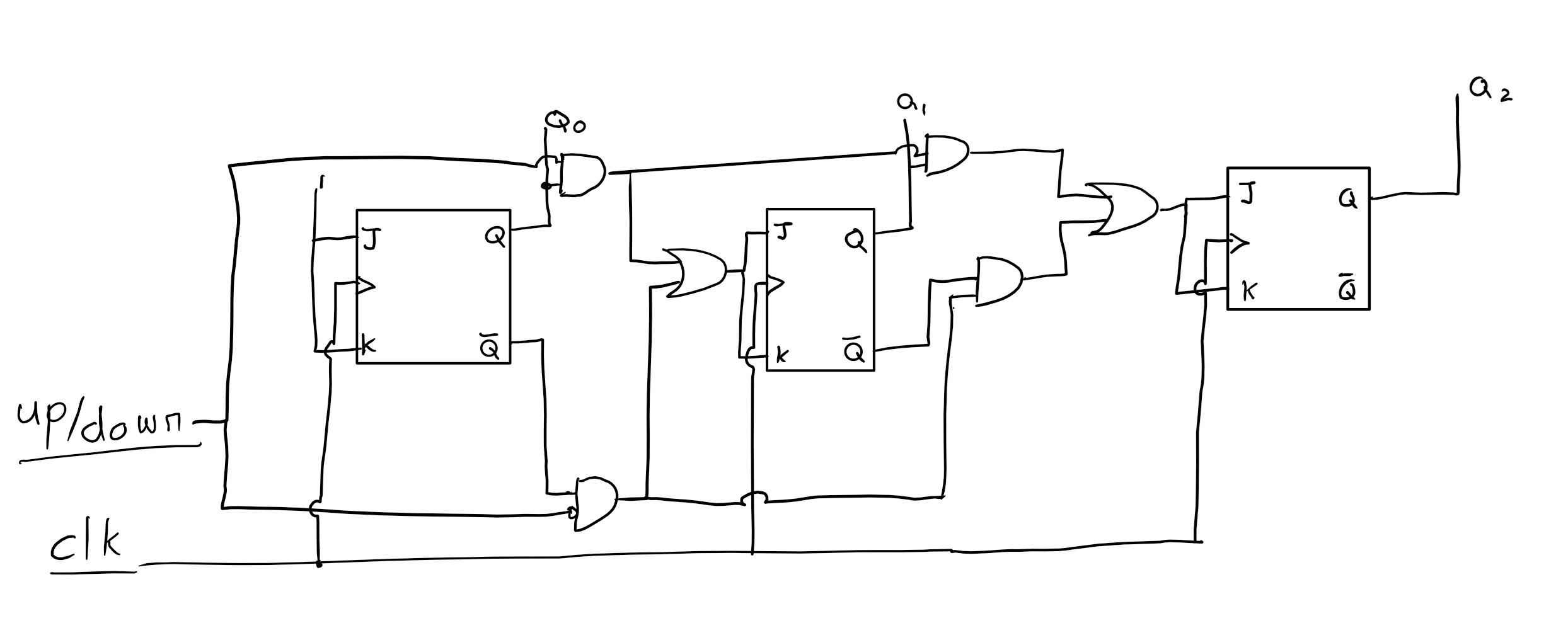
011. Last bit toggles, previous was 0 so second bit doesn't toggle

100. Etc

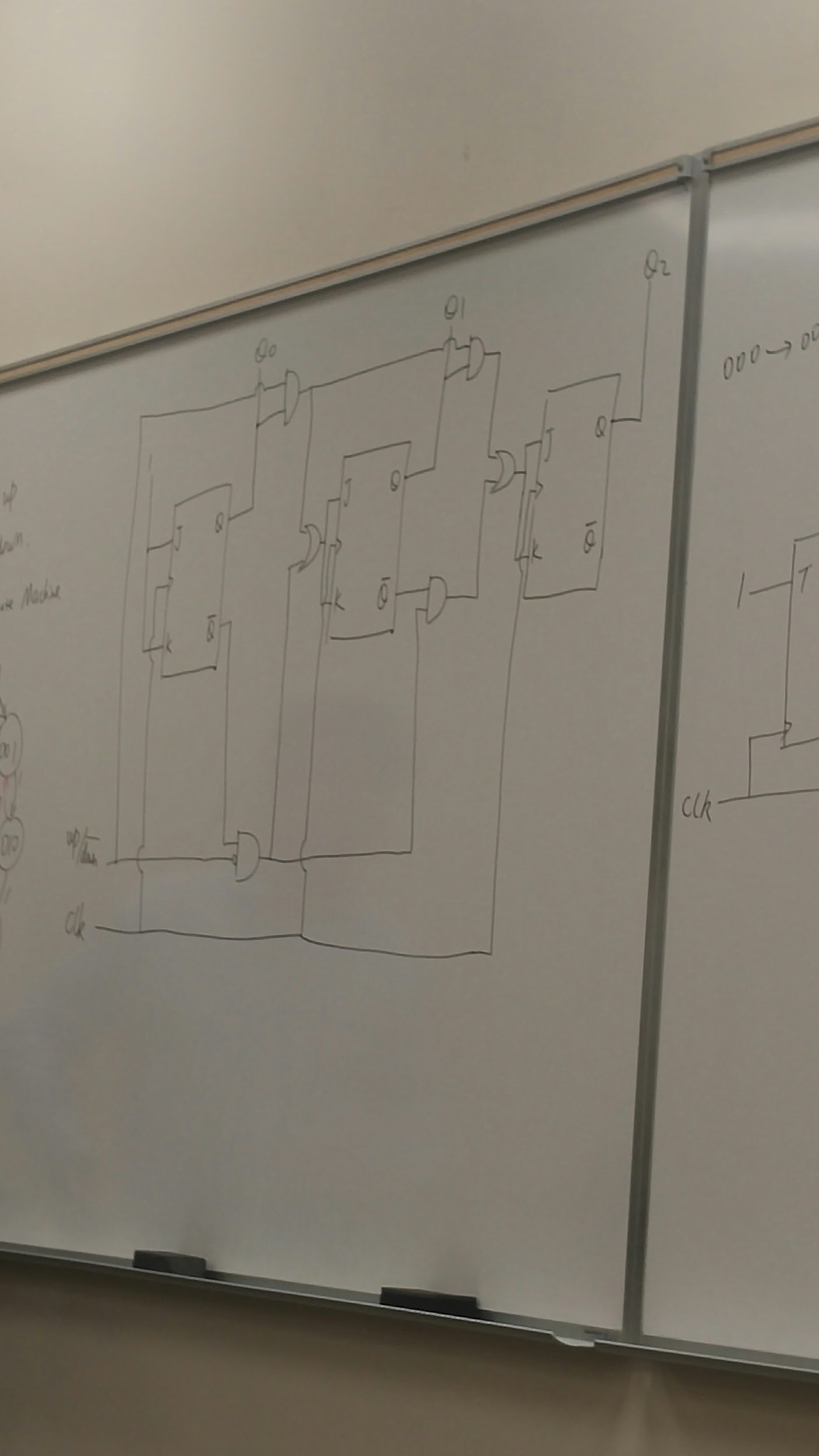
You can count up or down depending on the value of the first bit. (1=up, 0 = down)





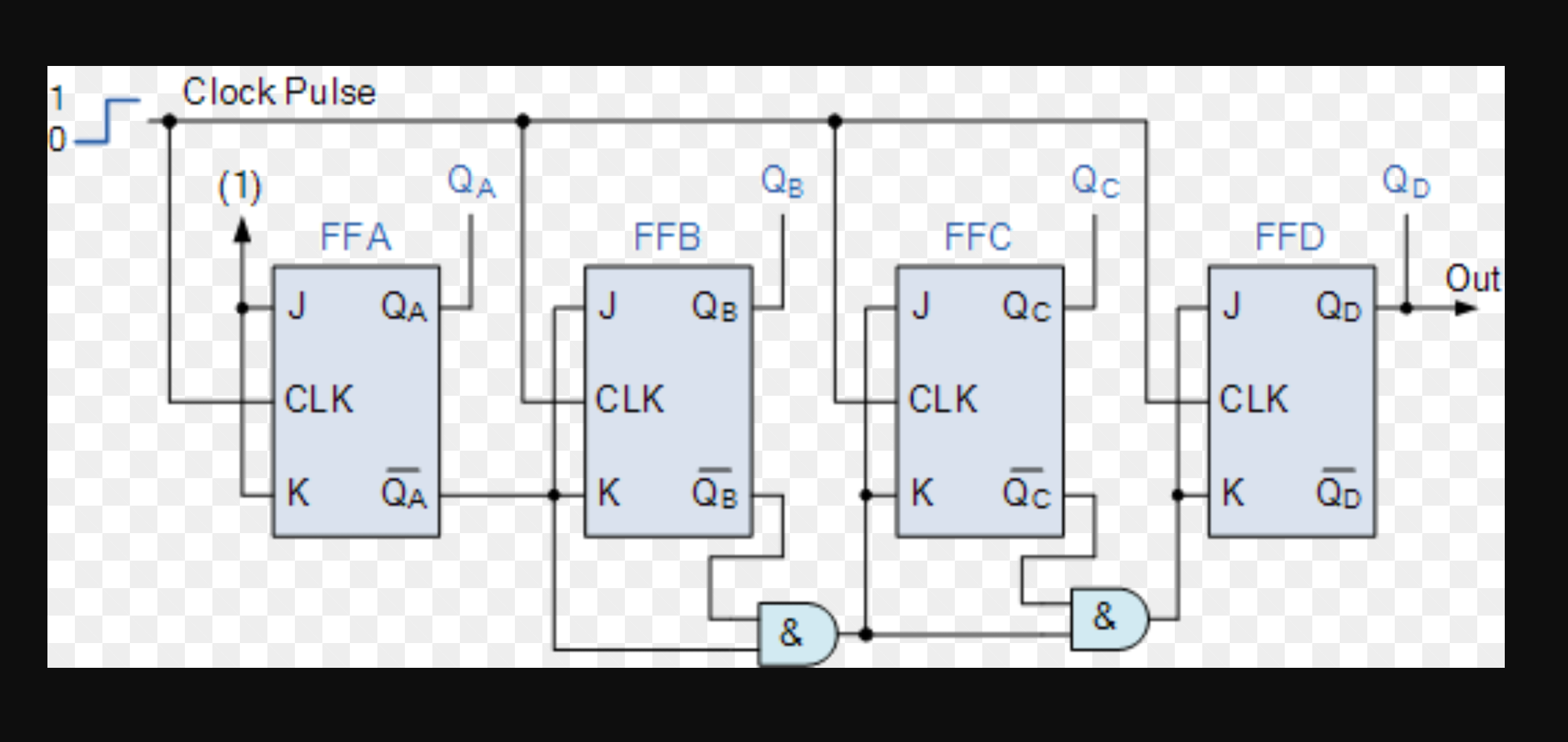


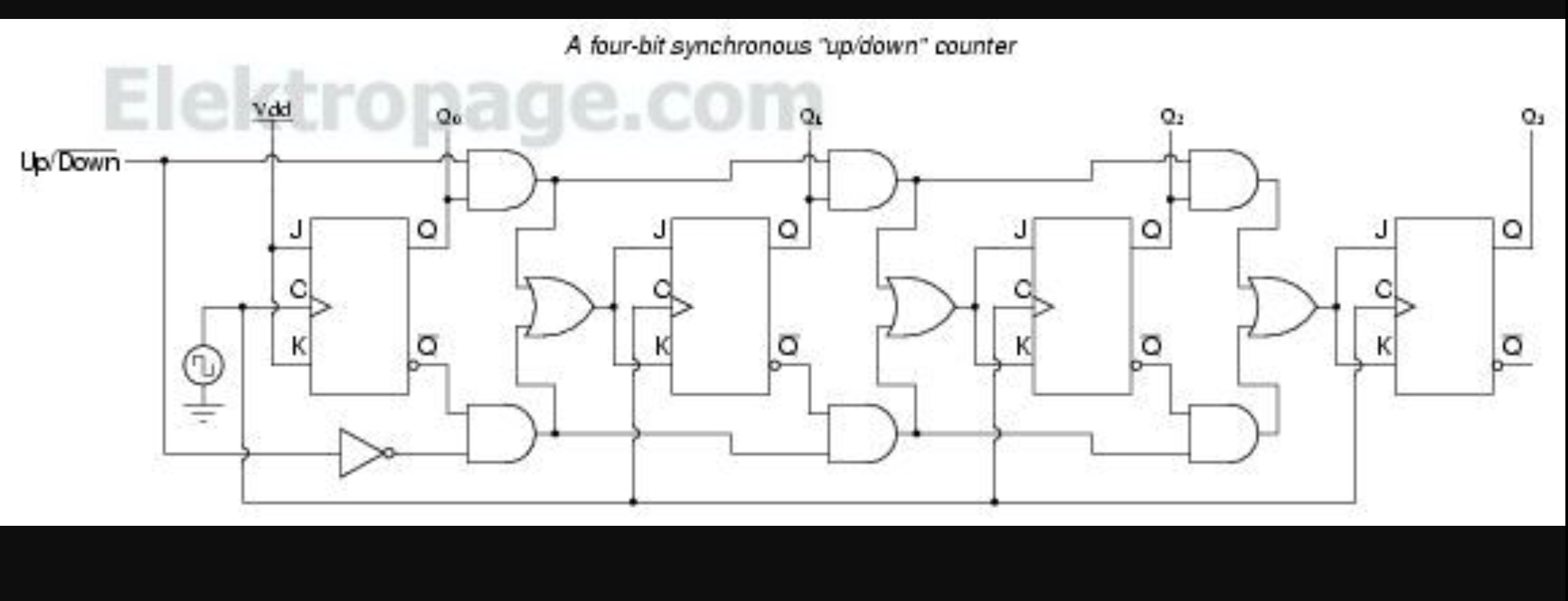
(Clearer image - Verify with image below. Note any discrepancies if they exist)

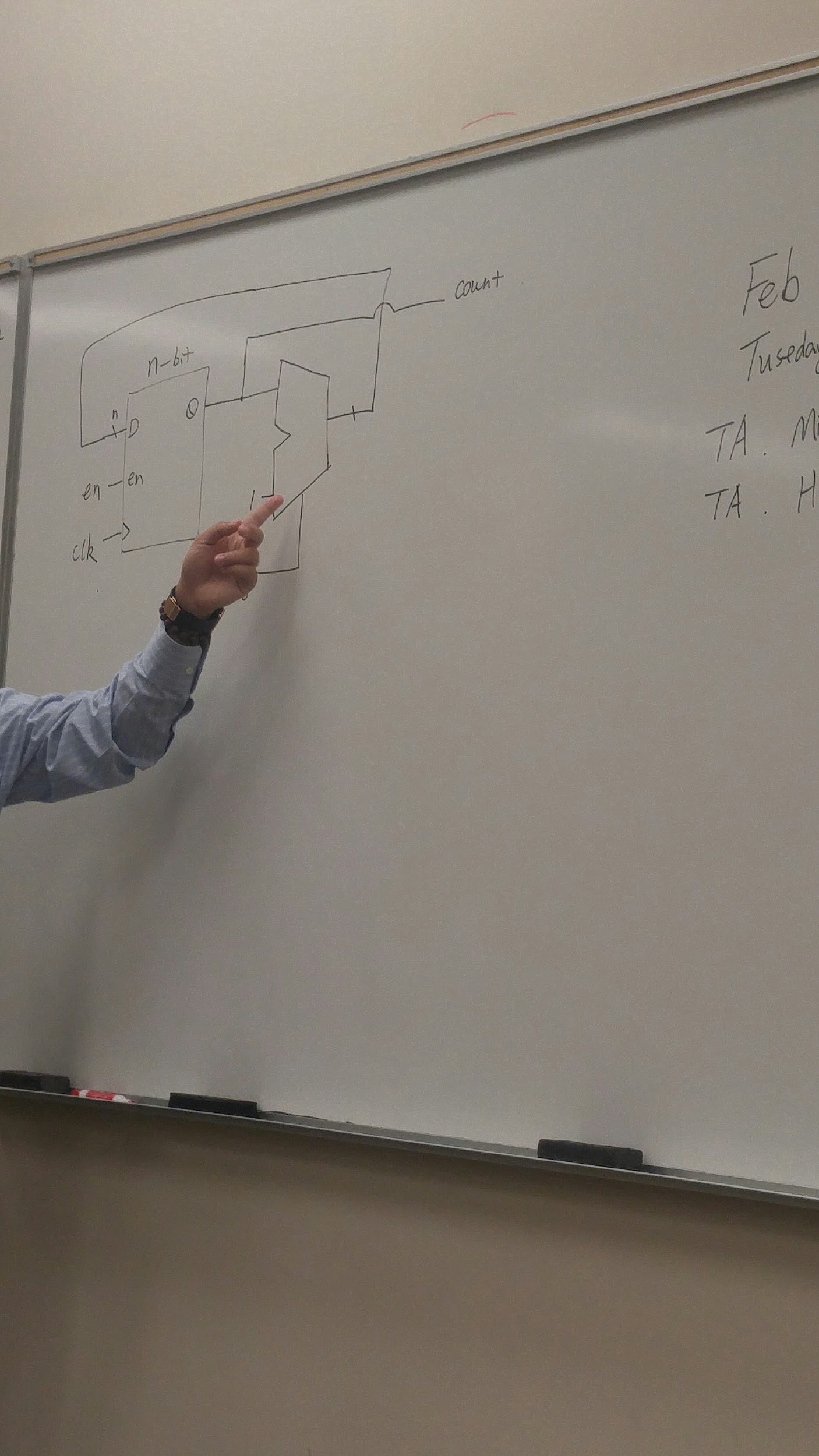


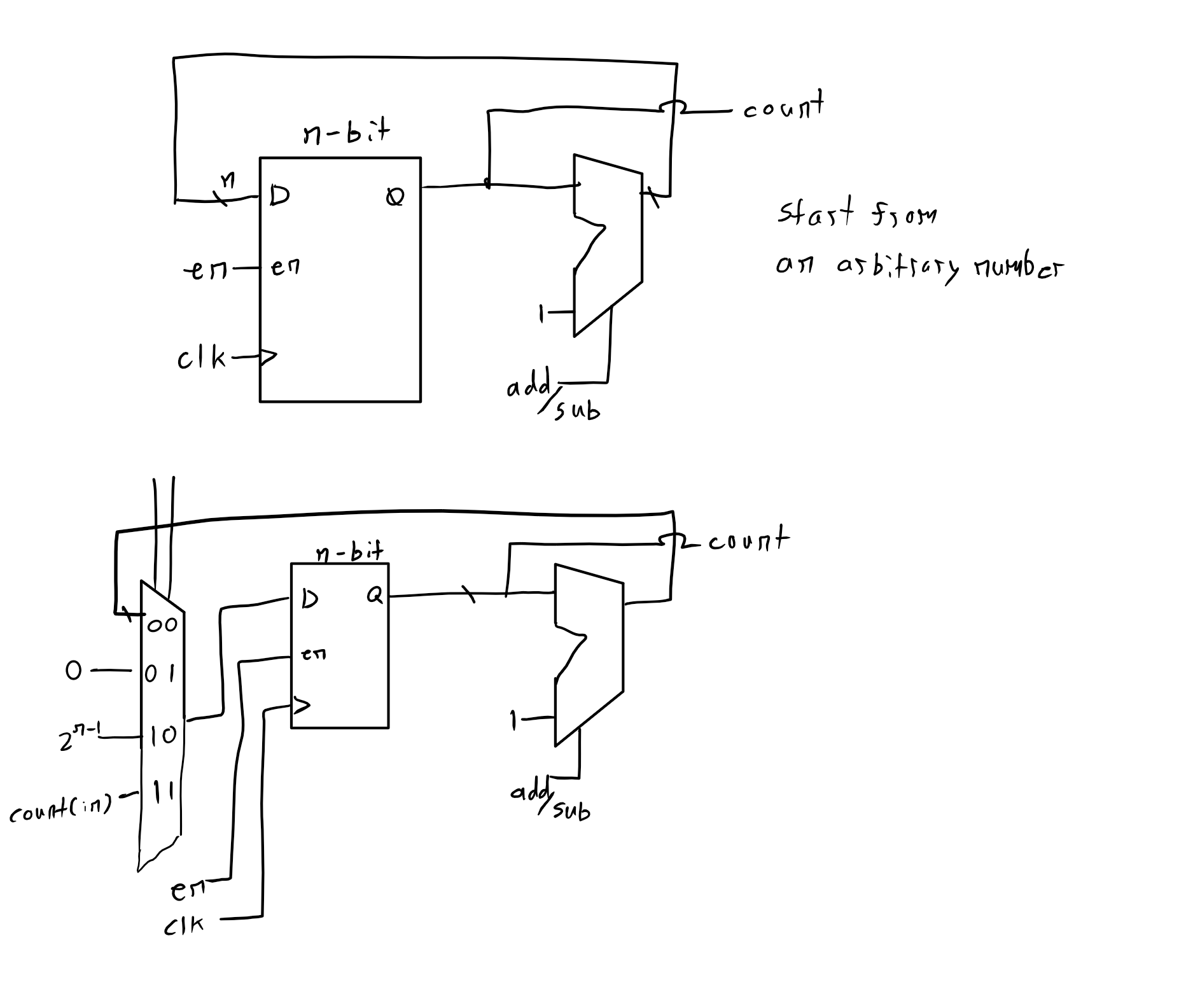
You can control the direction of the counter in this circuit.

Here's a prettier circuit:



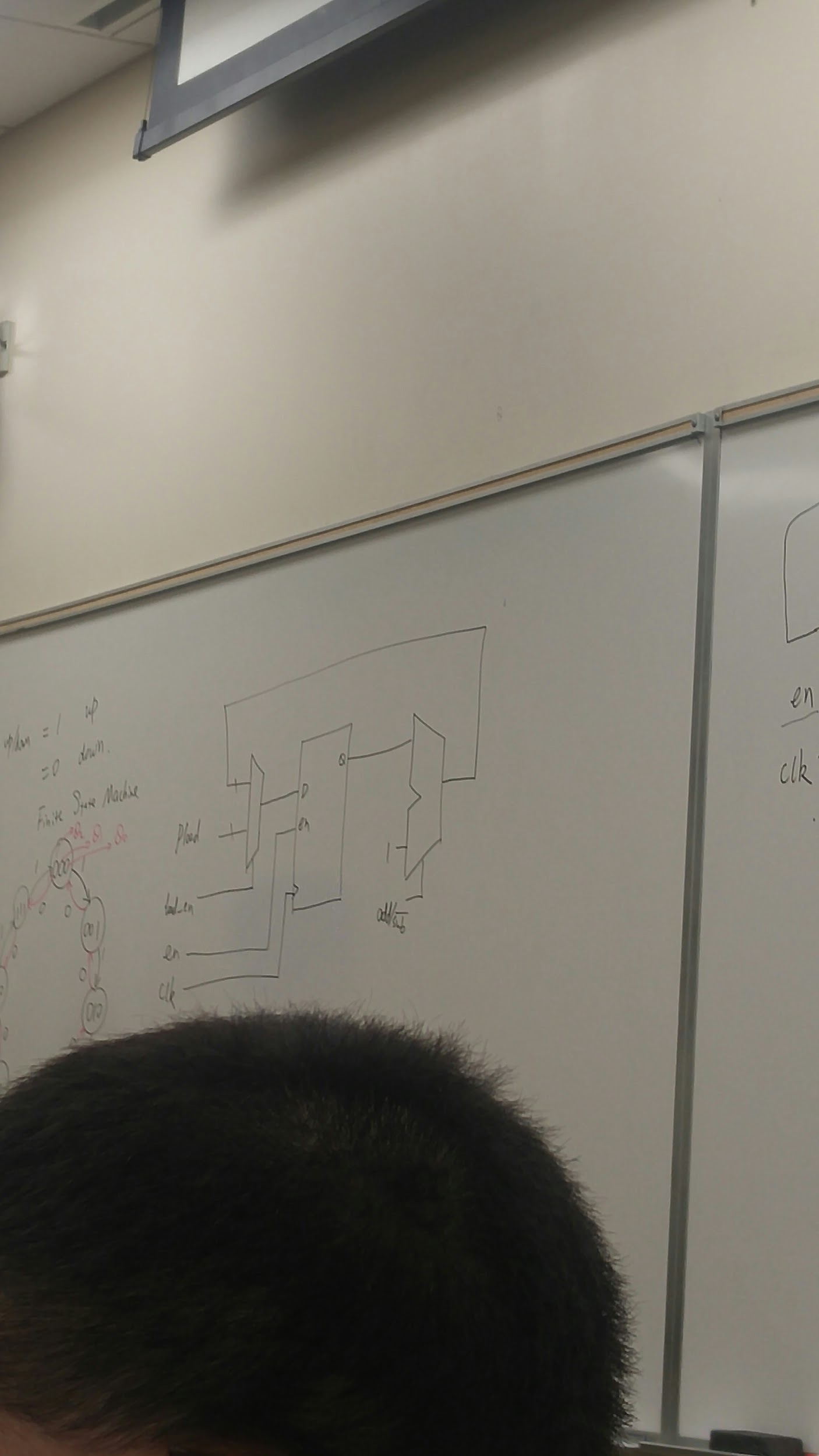






An ALU that adds or subtracts based on the signal. It is similar to a multiplexor in this case.

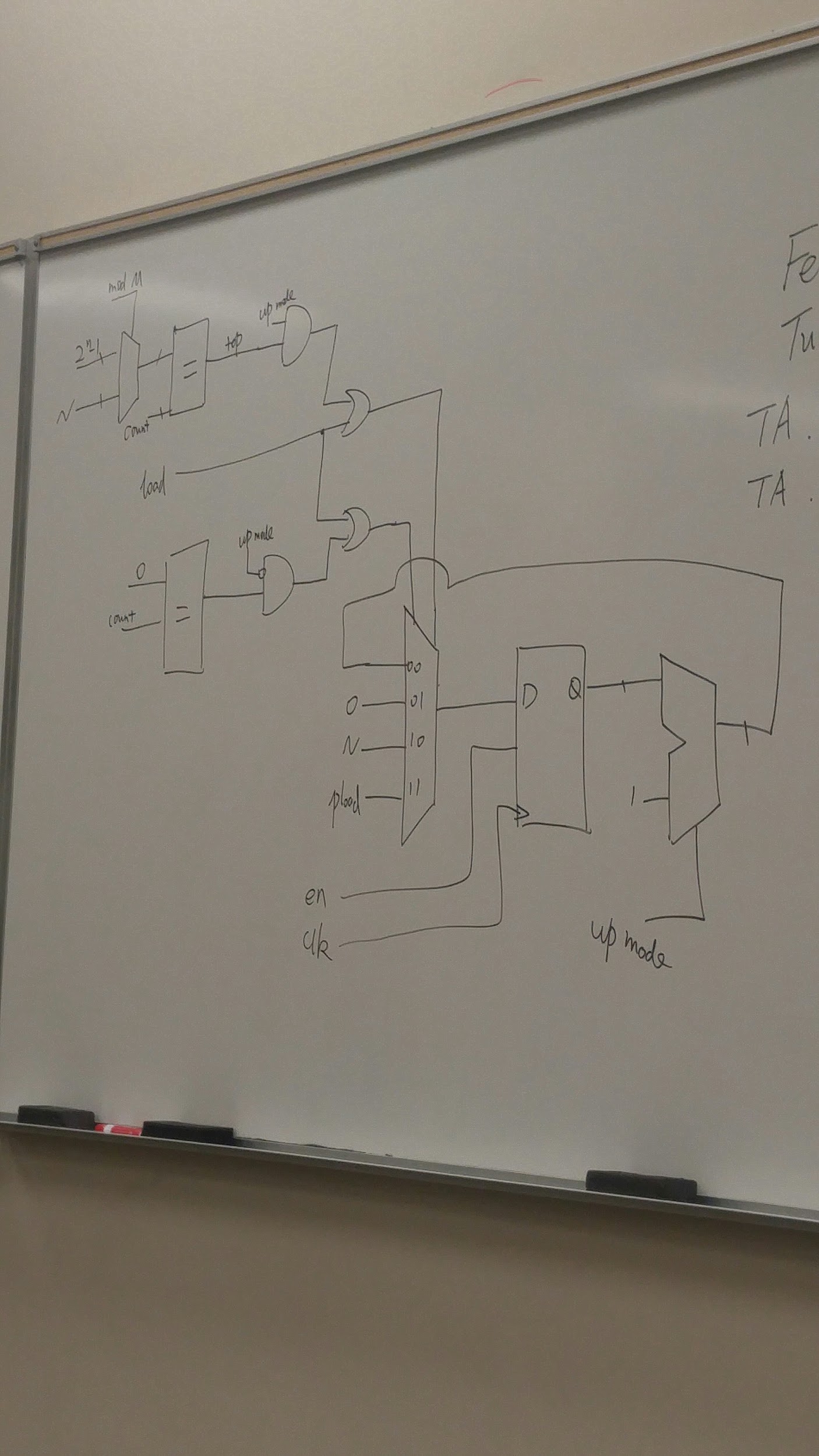
The multiplexor allows you to load a value to count from.

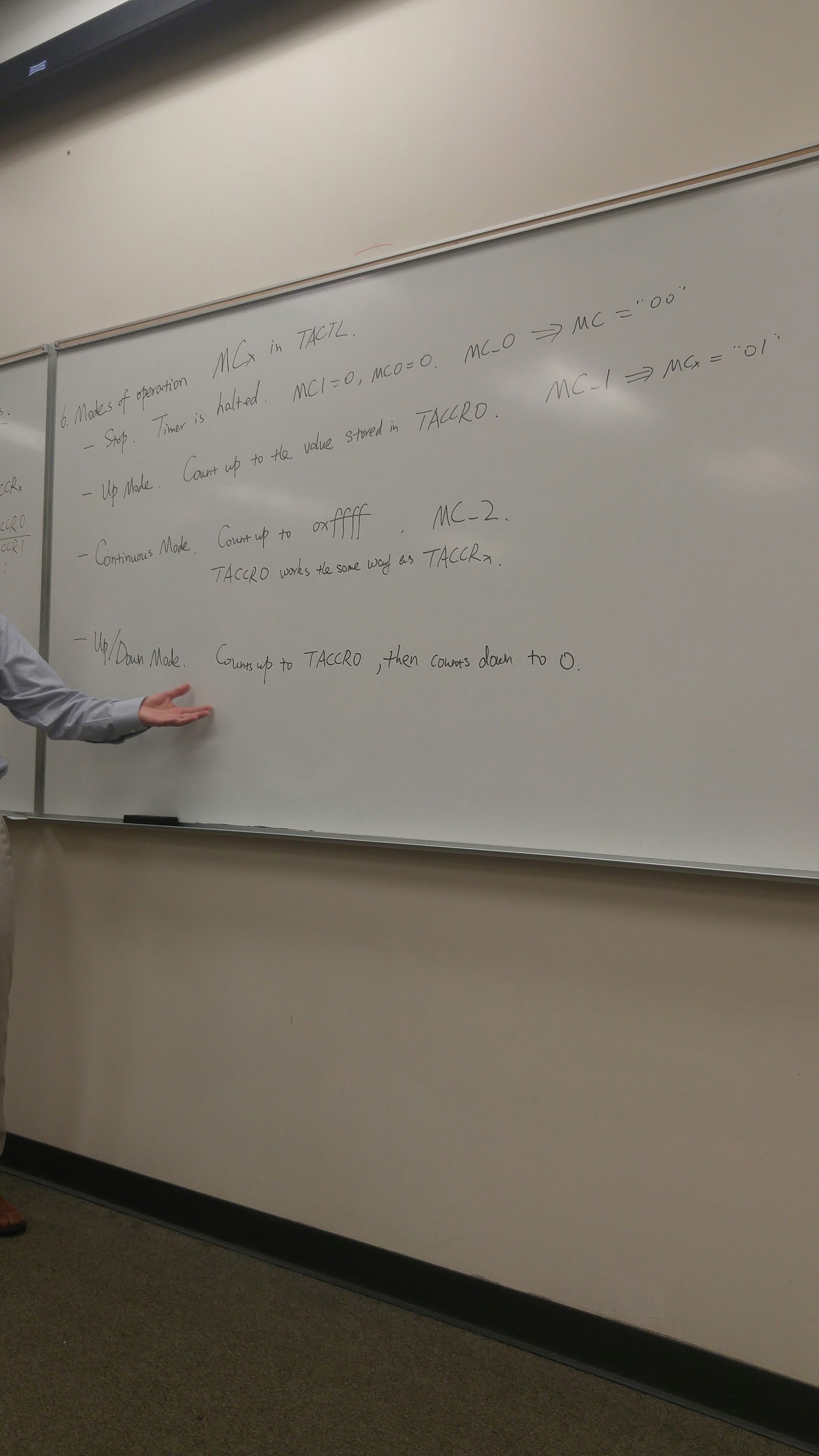


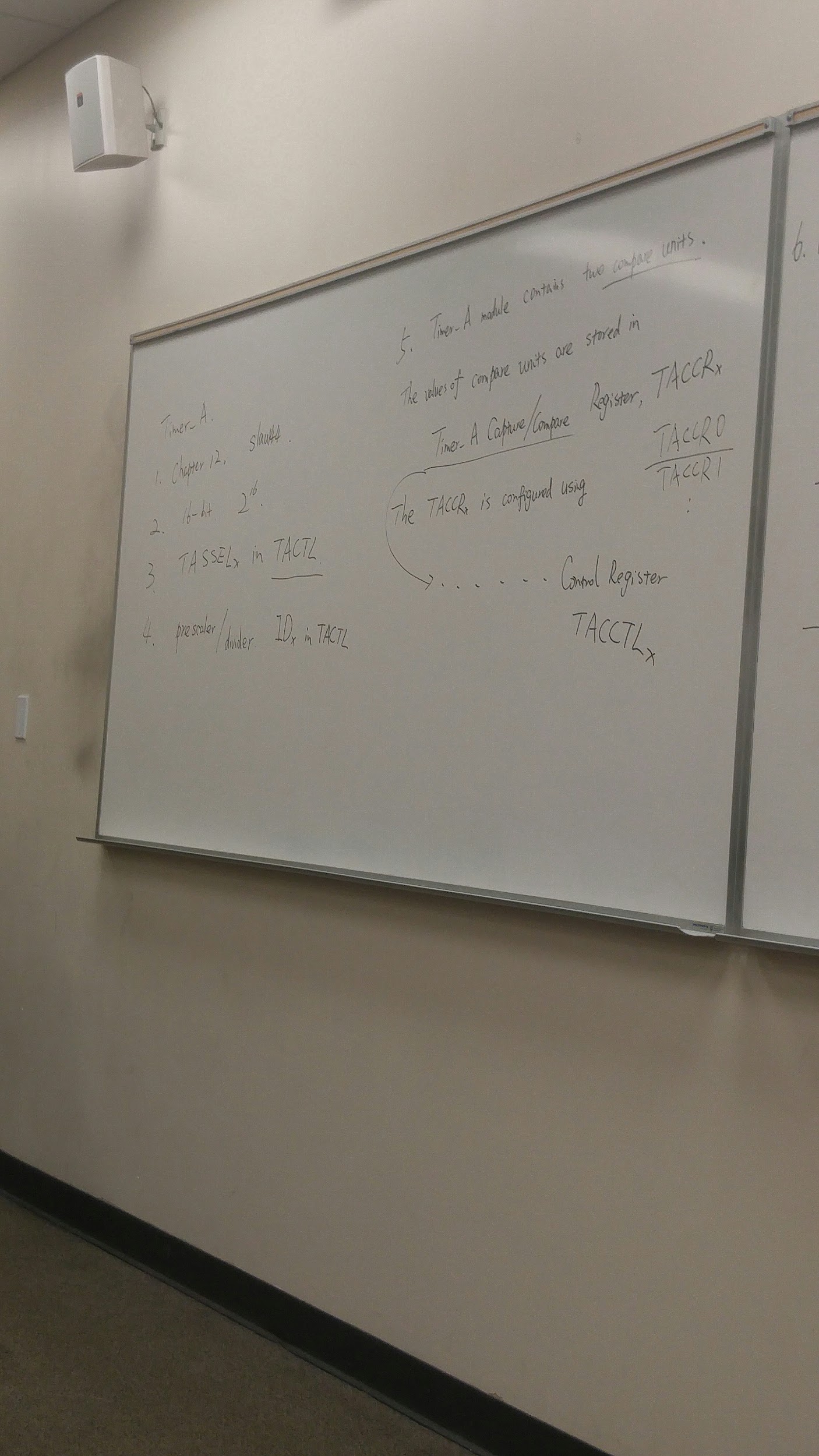
To count to any arbitrary number, you can compare that number with the current state of the counter.

Then, if the counter reaches that number, reset to zero.

This comparison is largely done by subtracting the arbitrary number from the current state of the counter and checking whether it is positive or negative.







Links:  
1. [ Programming reference for assembly ]

<http://ece.gmu.edu/coursewebpages/ECE/ECE447/F09/lab_docs/MSP430_Programming_Reference_r3.pdf>

2. [All about timers]

<http://www.ccs.neu.edu/home/noubir/Courses/CSU610/S07/MSP430-Clock-Timers.pdf>

3. [Voltages of inputs apparently]

<http://www.ti.com/lit/ds/slas735j/slas735j.pdf>

4. [More Timer notes, simple to understand]

<http://coecsl.ece.illinois.edu/me461/Lectures/ME461_L06_Timers.pdf>