COS10004 Computer Systems

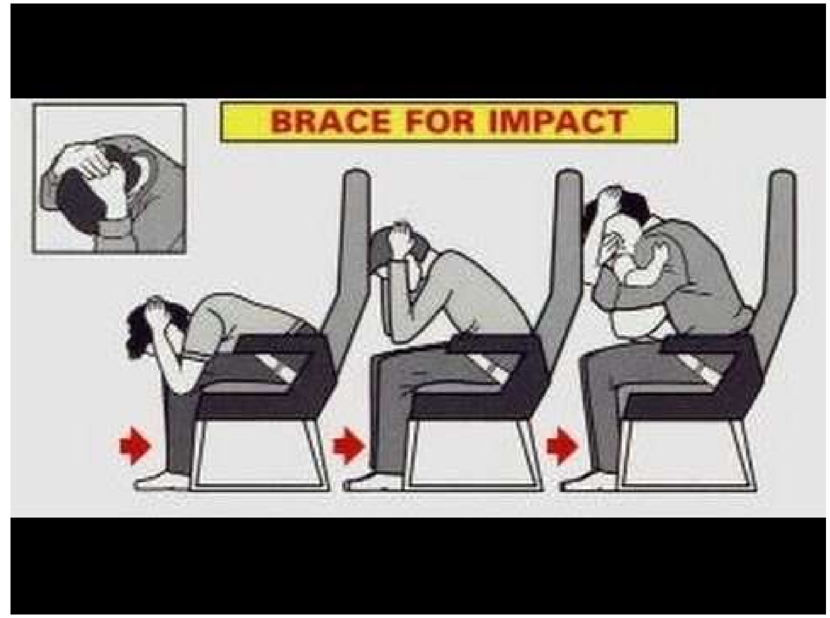
Lecture 12.2 - Unit Wrap Up

SWINBURNE UNIVERSITY OF TECHNOLOGY

CRICOS provider 00111D

You made it!

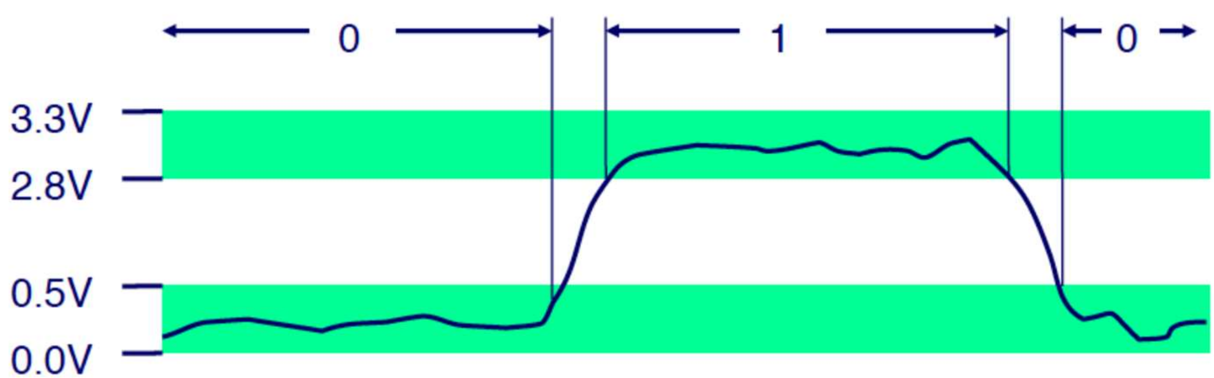
Gimme feedback!



- Your Unit. Your Say.
  - Feedback is critical
  - High response rates make sure we get an accurate picture
- In fact .. You can do it now!

Information and Computers

- What is information?
- Computing requirements to process information: representation, manipulation, storage
- Binary information: two states (on-off, true-false)
- Bit (Binary digit) notation: 0 and 1



Number Systems

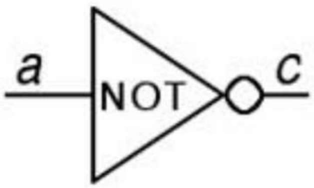
- Modern computers operate with digital representations of information:
  - Easier to work with but has implications
- Number systems and conversions to know:
  - Binary ↔ Decimal
  - Hex ↔ Binary
  - Hex ↔ Decimal
  - And the formula in general

Number Systems

Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

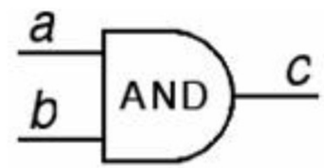
Hex	Binary
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Gates



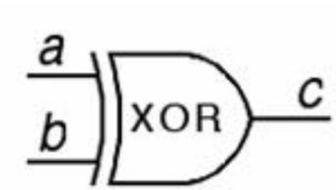
$c = \bar{a}$

a	c
0	1
1	0



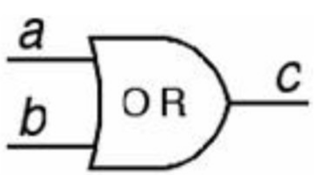
$c = a.b$

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1



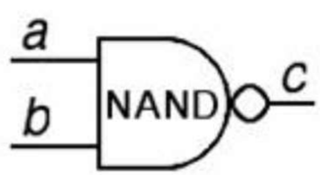
$c = a \oplus b$

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0



$c = a + b$

a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

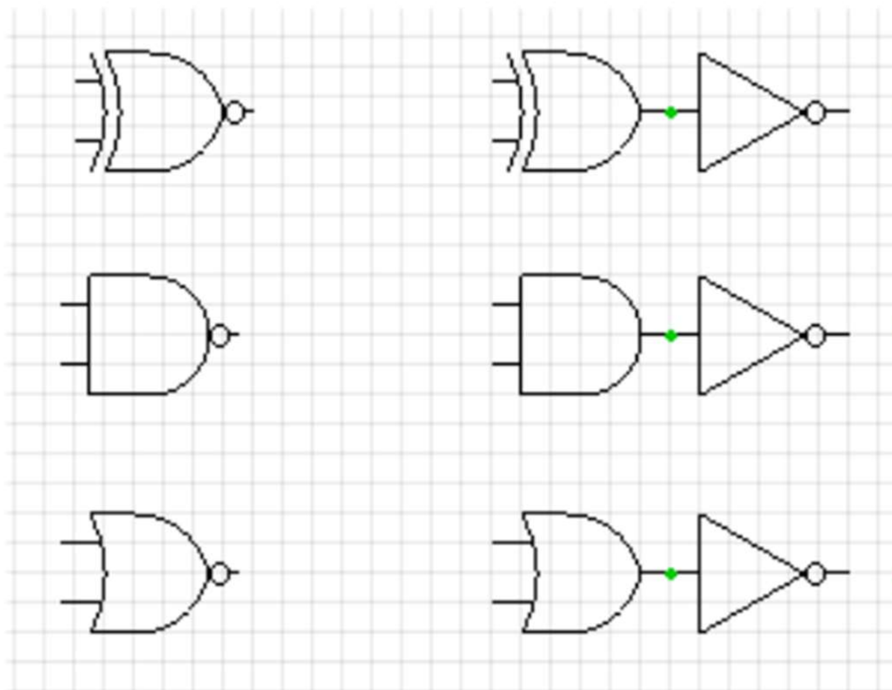


$c = \overline{a.b}$

a	b	c
0	0	1
0	1	1
1	0	1
1	1	0

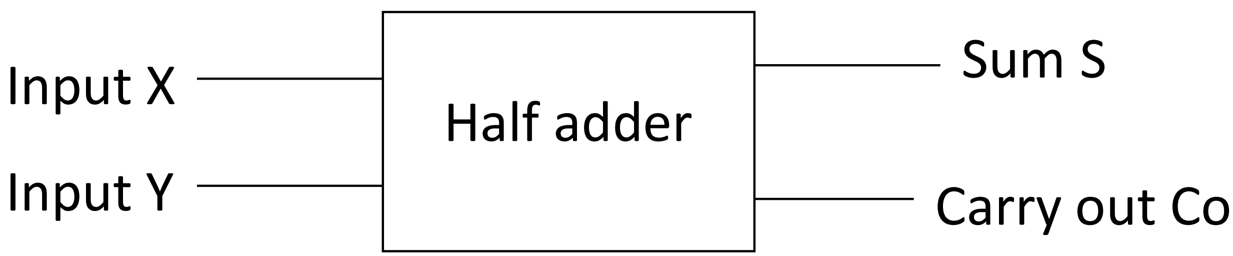
Gates

- An inverter can be added to the output of any gate to reverse it's output (making an N version)
- NXOR
- NAND
- NOR



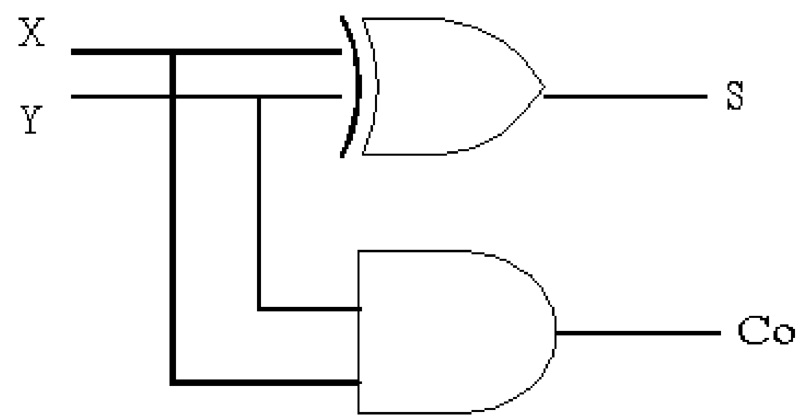
- !^
- !&
- !|

Half Adder



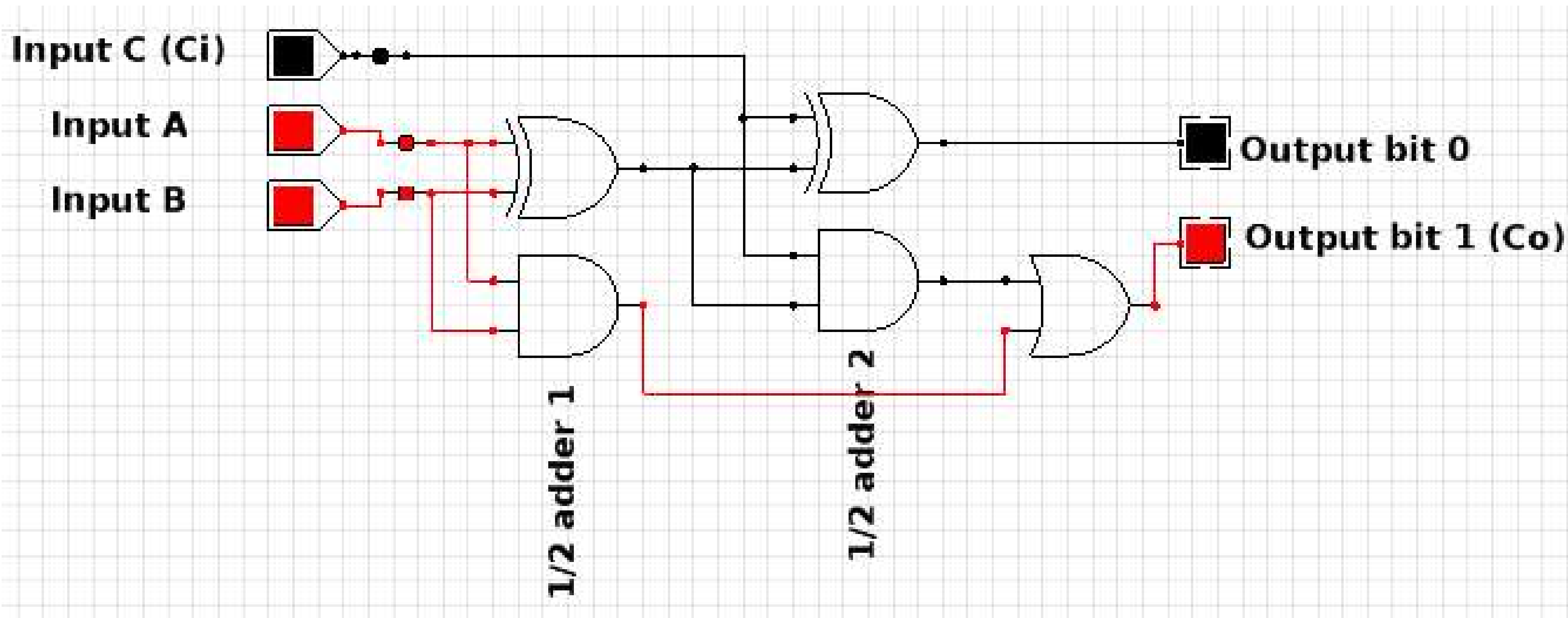
X	Y	S	Co
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$\boxed{S = X \oplus Y}$ $\boxed{Co = X \bullet Y}$



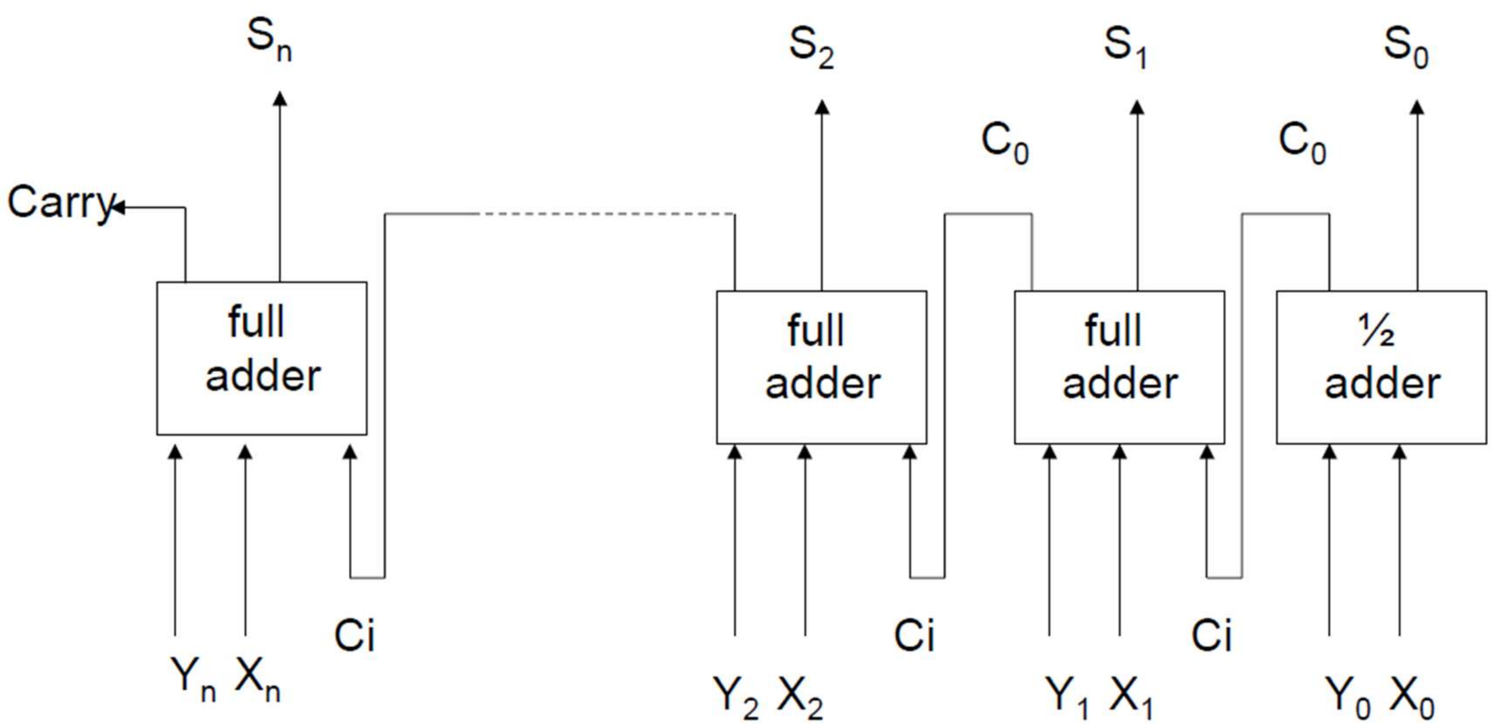
Full Adder

- We could make this from two half-adders:

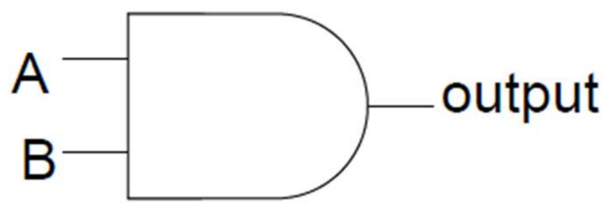


Adding more bits

- To add real numbers together (8, 16, 32 bits...) we need to cascade full adders together.



Programmable gates

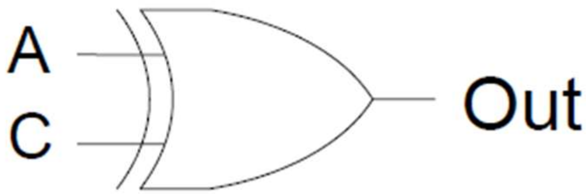


A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

If A=0, output always 0

If A=1, output = B

Programmable gates

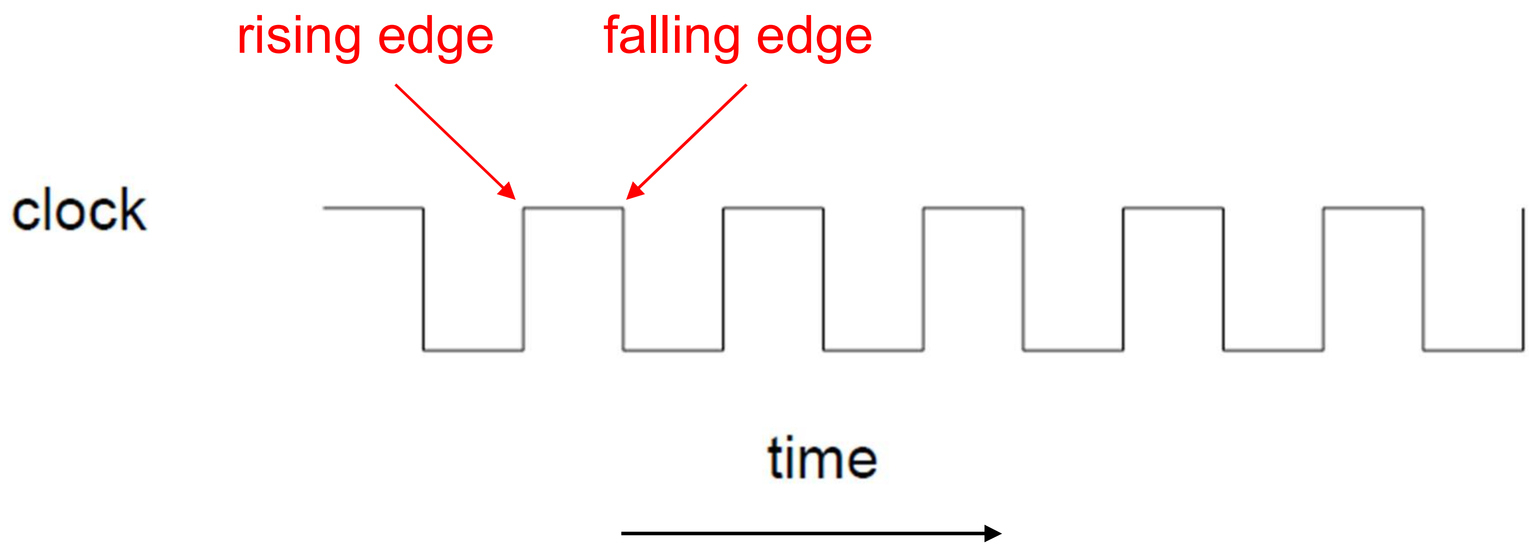


C	A	Out
0	0	0
0	1	1
1	0	1
1	1	0

If C=0, output = A

If C= 1, output = $\overline{A}$

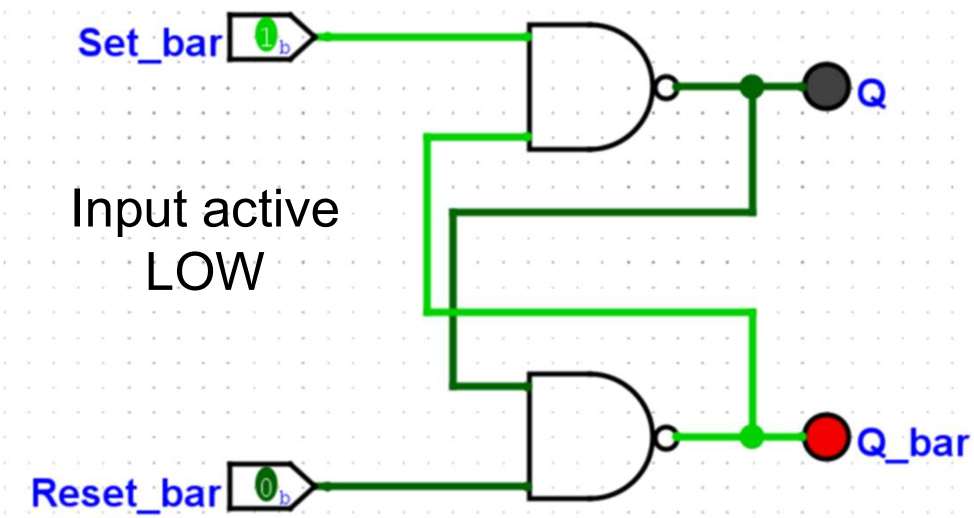
Clock feeds into the ALU

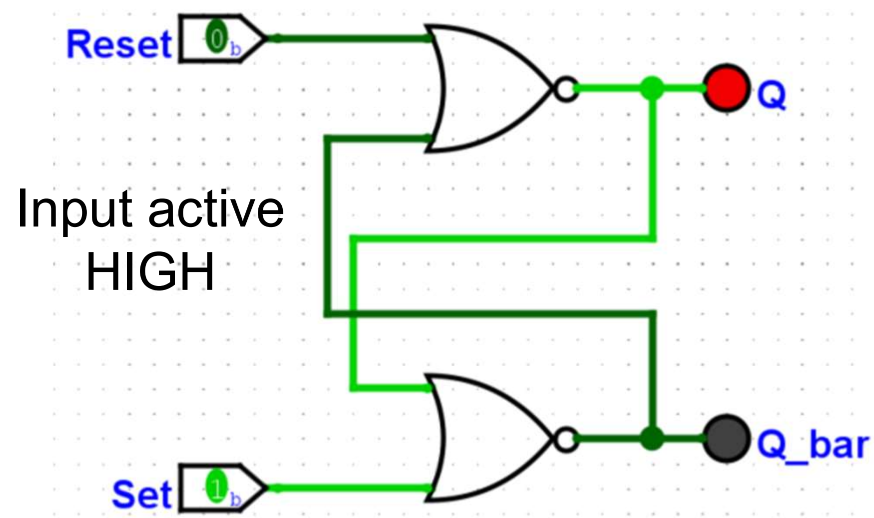


The *clock* is needed because bits need to “settle” before you can use them.

Computers often have different clocks controlling different parts.

RS Flip-Flop



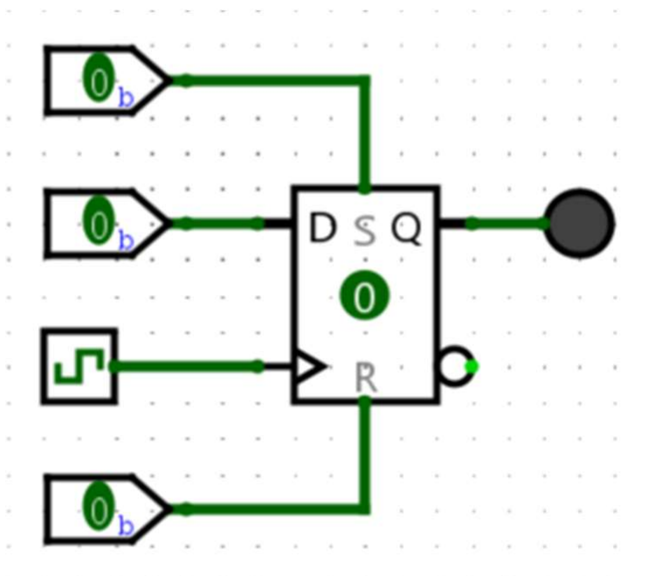


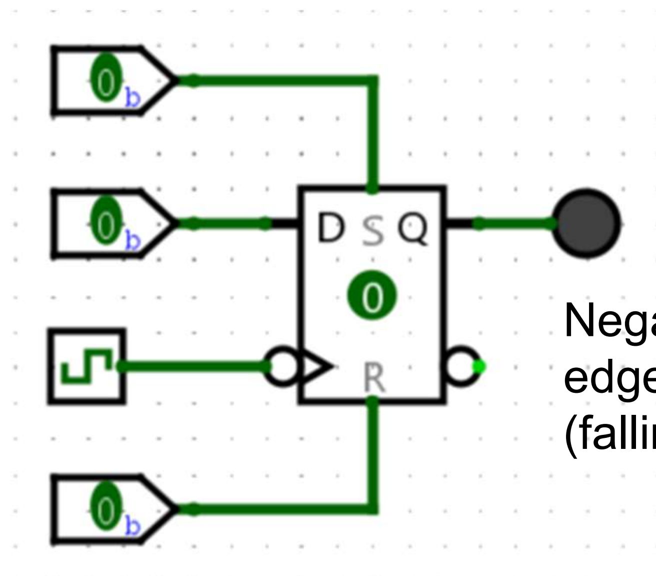
$\overline{SET}$	$\overline{RESET}$	Q	$\overline{Q}$
0	0	indeterminant dangerous!!!	
0	1	1	0
1	0	0	1
1	1	no change	

SET	RESET	Q	$\overline{Q}$
0	0	no change	
0	1	0	1
1	0	1	0
1	1	indeterminate	

Clocked Flip-Flops – D Flip-Flop

Positive
edge-triggered
(rising edge)





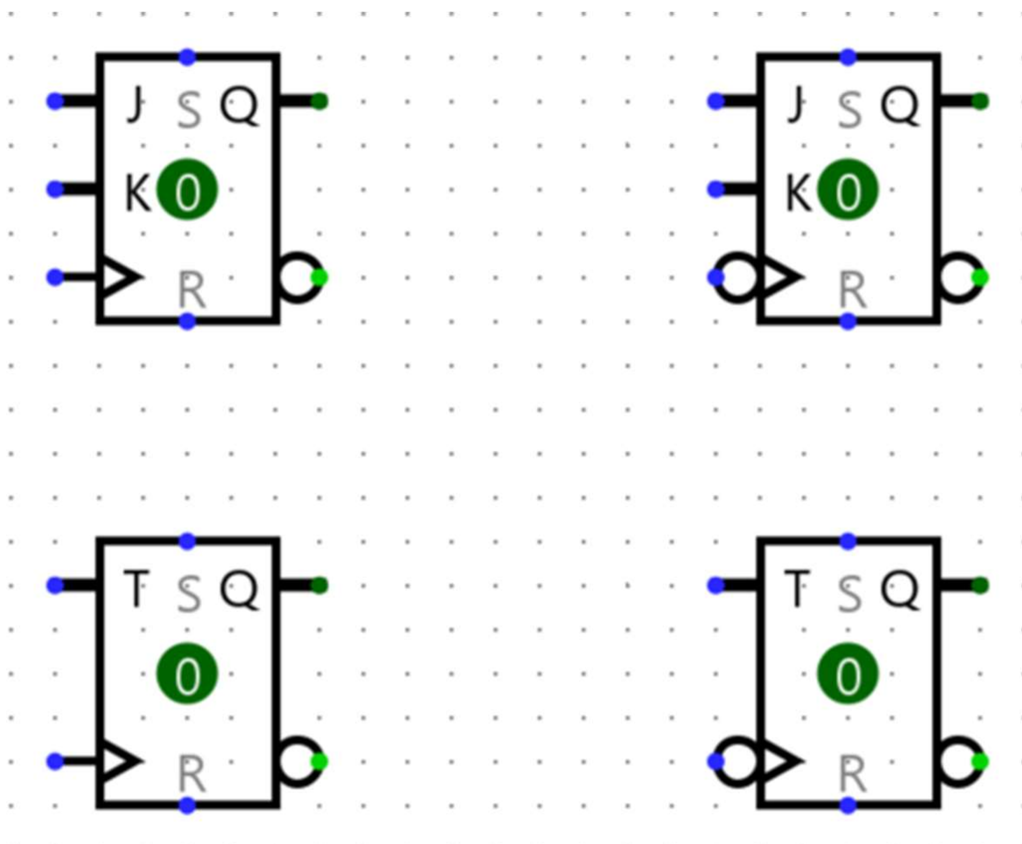
Negative
edge-triggered
(falling edge)

D_N	Q_{N+1}
0	0
1	1

(N means at the clock, N+1 means after the clock)

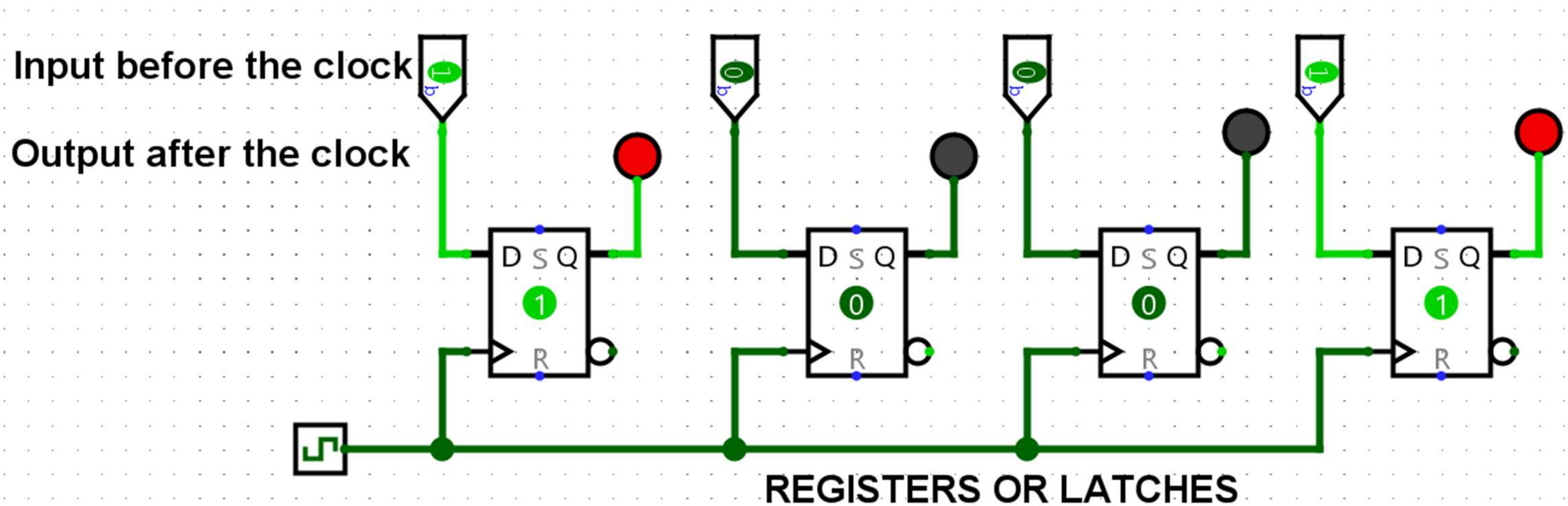
D-FFs are used in computer registers and memories and in counters and shift registers

Clocked Flip-Flops – JK Flip-Flop



J	K	Q_{N+1}	
0	0	Q_N	(No Change)
0	1	0	(Reset)
1	0	1	(Set)
1	1	$\overline{Q}_N$	(Toggle)

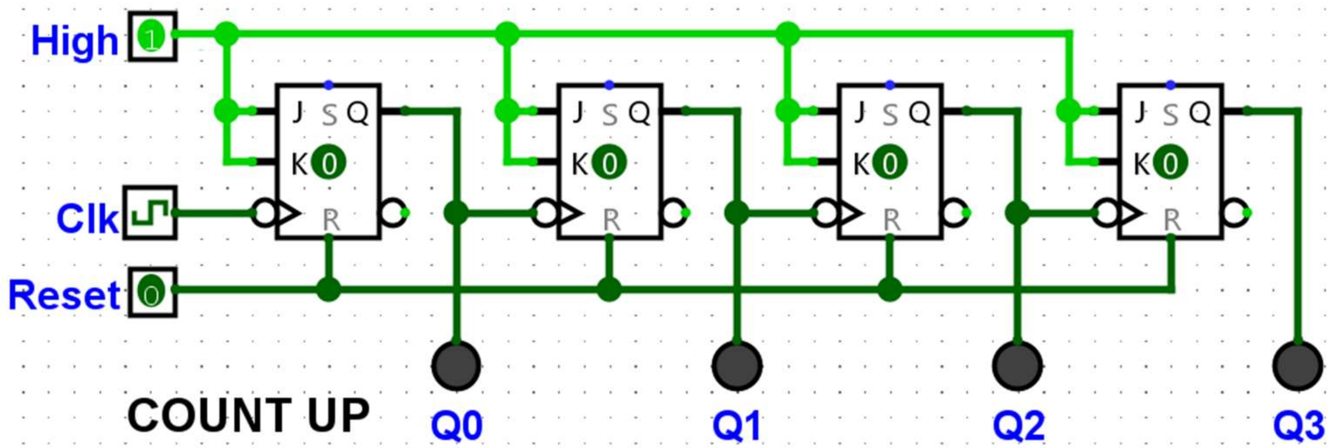
Registers (or Latches)



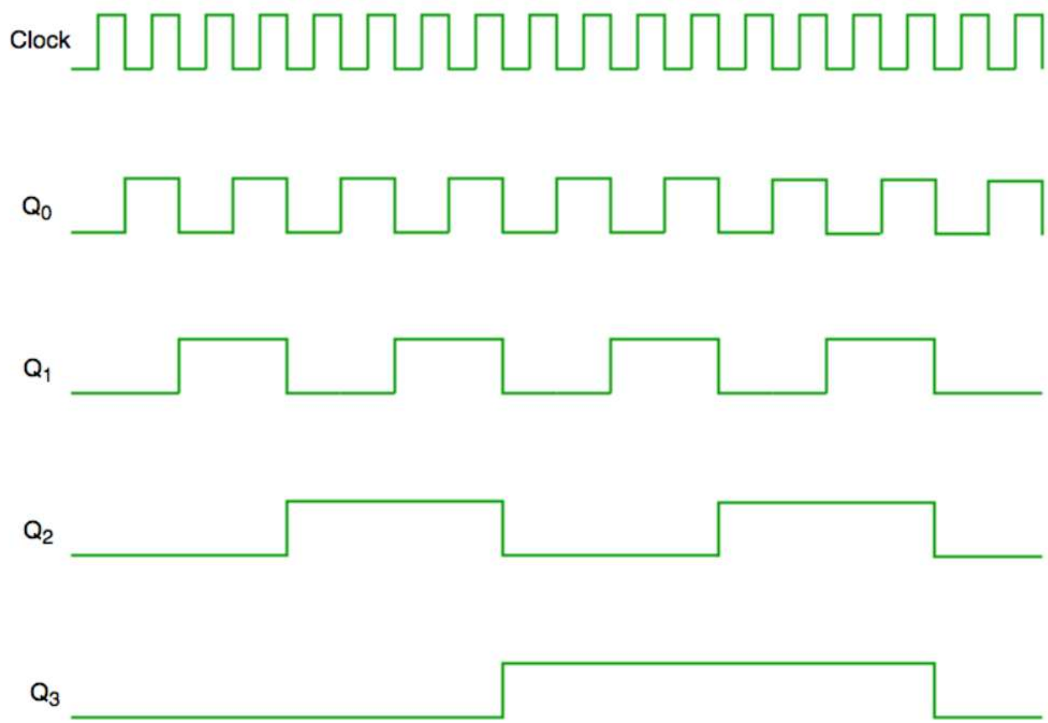
BIT Endianness

- Bits generally don't have an address, so definitions refer to the positional order of bits
- Big Endian:
  - The most significant bit (MSB) comes first
- Little Endian:
  - The least significant bit (LSB) comes first
- This matters for interpreting the value of a bit string (especially if bits are received as a serial stream!)
  - E.g., what is 1011 in decimal ?
  - 11_{10} (big endian), or 13_{10} (little endian)

Ripple Counter

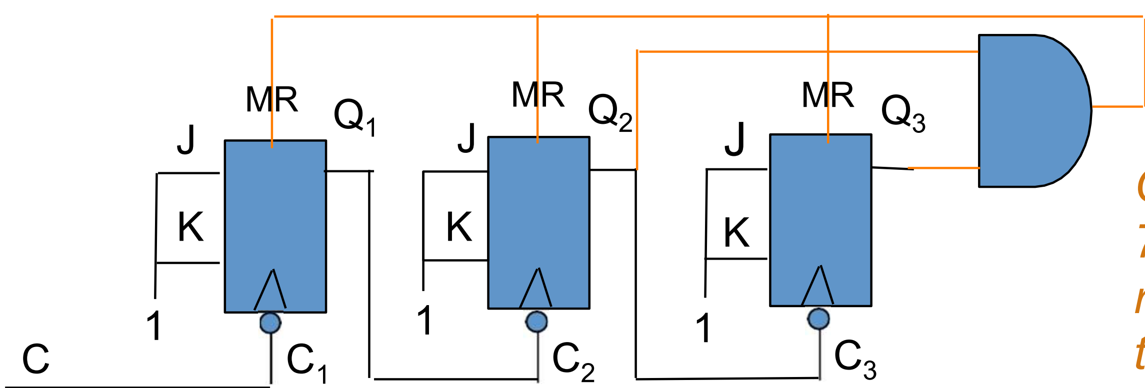


- Ripple counters utilise the toggle setting of J-K Flip Flops



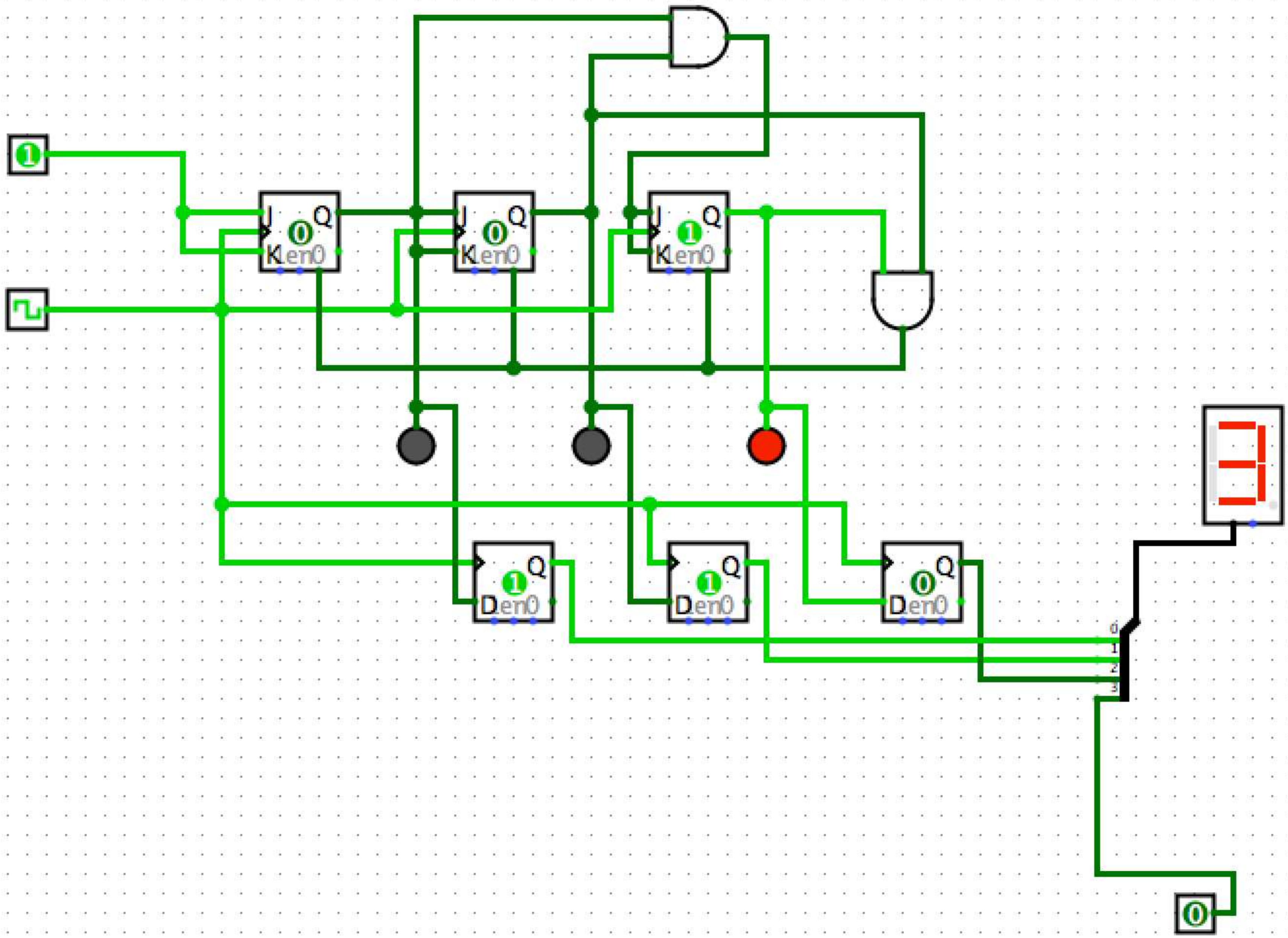
Modulo 6 counter with a momentary illegal state

- Detecting the first illegal state (6 in this case) and immediately resetting to 0 (don't wait for the clock) by using the asynchronous master reset (MR) or CLR'
  - This circuit uses a *cascading clock*



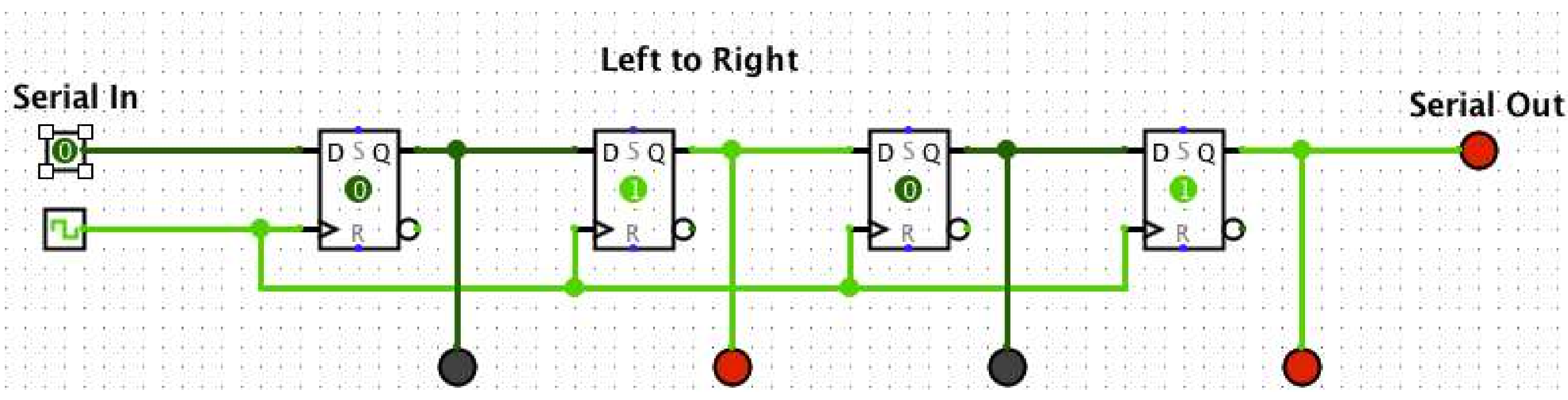
Goes 1 if counter on 6 or 7 forcing the master resets to clear each F/ F to zero

3 bit Common Clock - Little Endian - No illegal state



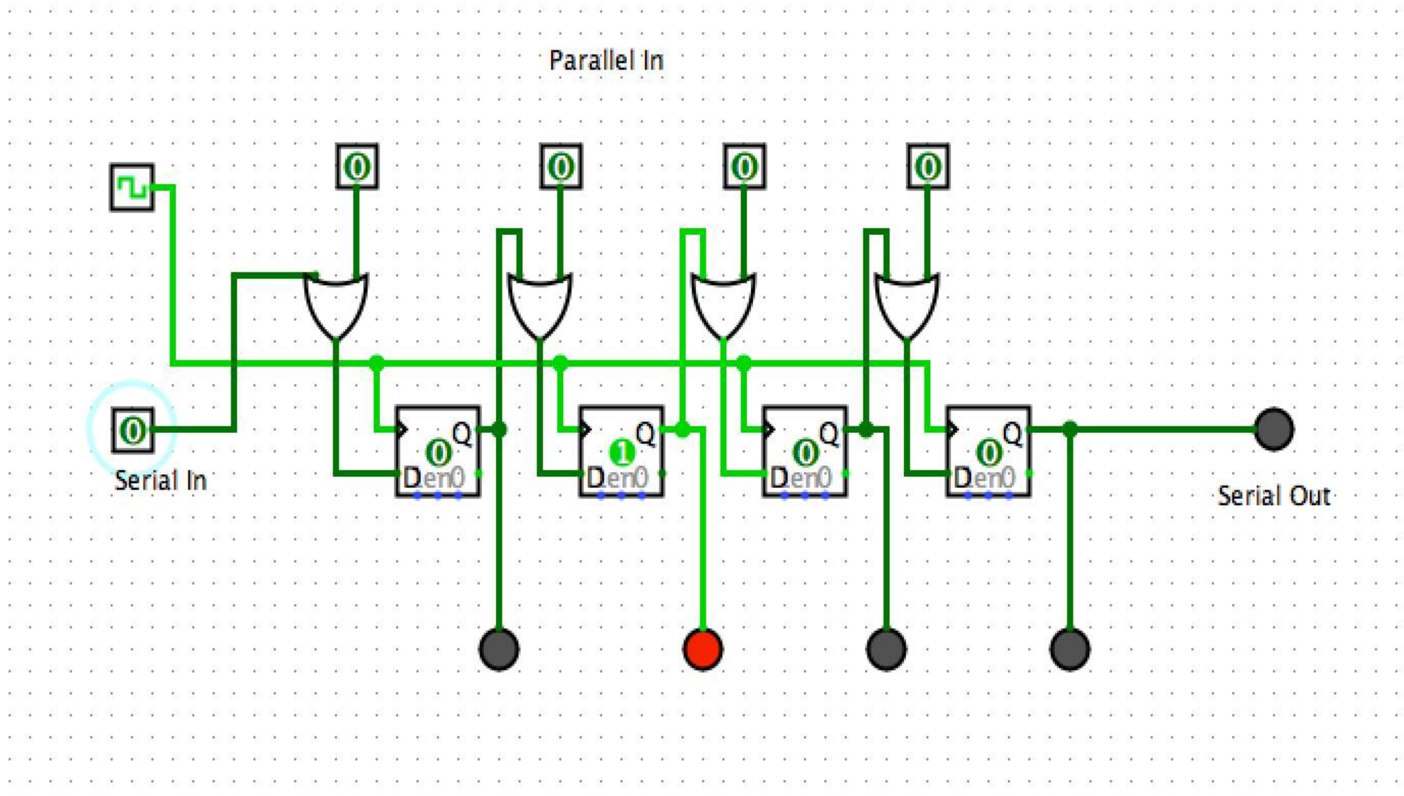
Shift Registers

- A shift register takes input from one end, and at each clock change this value is moved to the next D-Flip-Flop.
- This is used in serial data transfer when a byte (say) of data sent on a cable one bit after another can be collected in a series of D Flip-Flops to rebuild the whole data byte. This is called ***serial-to-parallel*** conversion.



Serial-to-Parallel Conversion

- Some shift registers allow all flip-flops to load at once, i.e., in *parallel.*
- This gives ***parallel-to-serial*** conversion



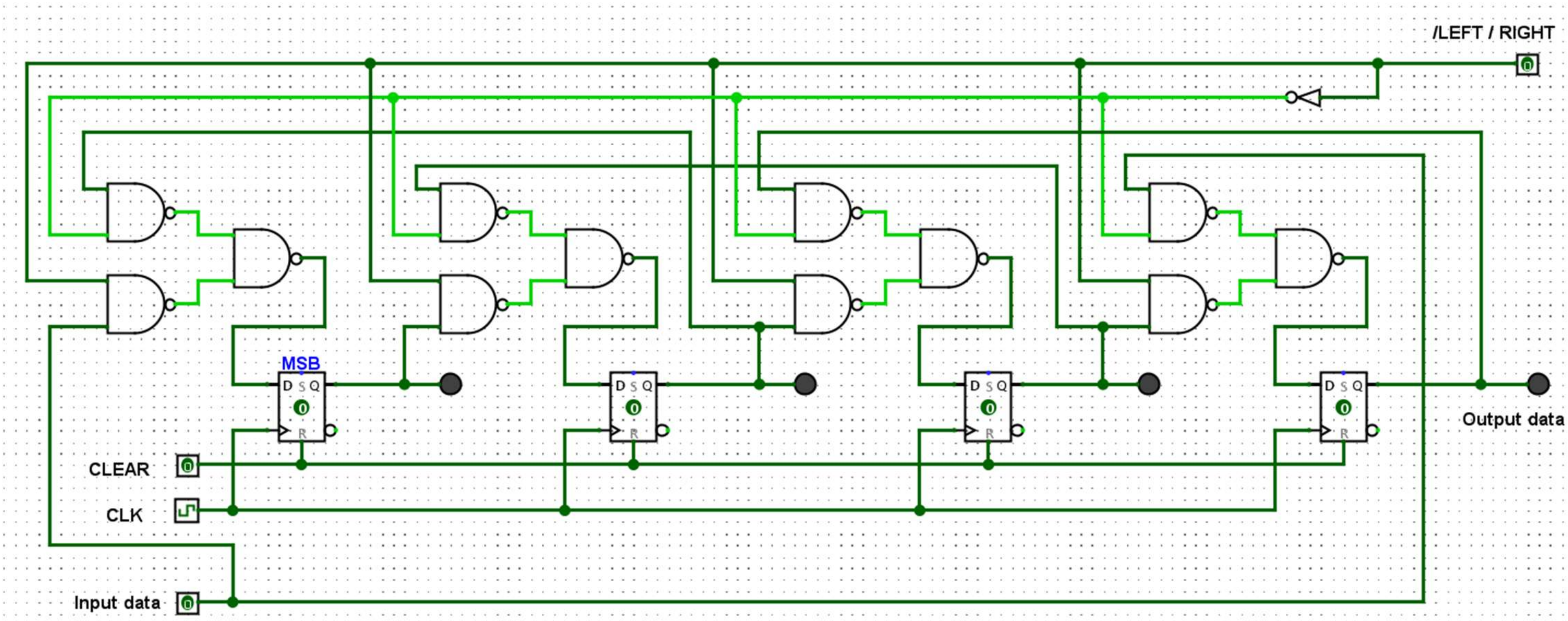
Memory

- Many different types of memory:
  - ROM variants: PROM, EPROM, EEPROM/Flash, …
  - RAM variants: SRAM, DRAM, SDRAM, DDR SDRAM, …
  - Trade offs of speed, space, expense, longevity
- All slower to access than registers
- Memory addressing:
  - Bits need to address each individual byte
  - m-bit address bus $\rightarrow$ 2^m addressable locations

Stacks

- Random access memory requires knowing the address of every byte/word you want to access
- Stacks offer a way of organising and accessing memory without random (indexed) access:
  - There are hardware stacks and software stacks.
- Hardware stacks created out of dedicated shift registers
- Software stacks typically defined in RAM using conventions

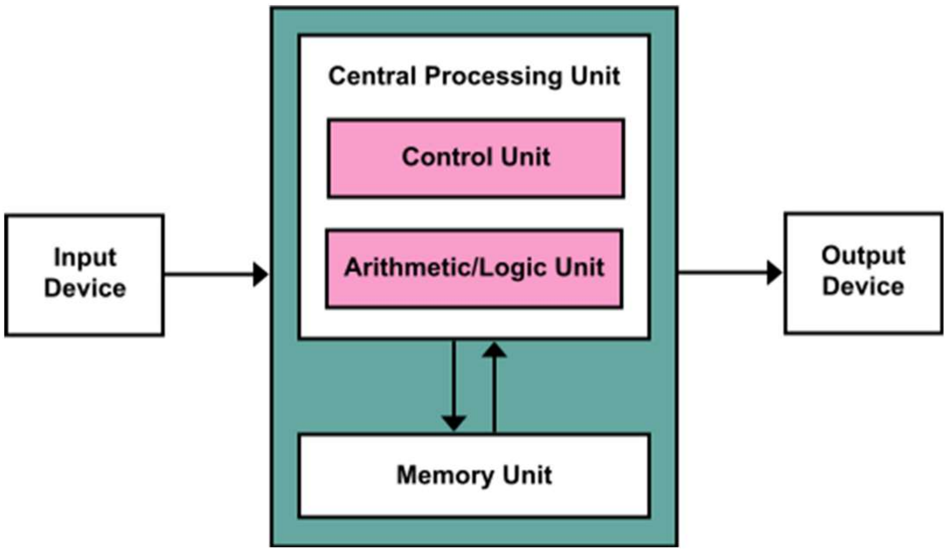
4-bit depth hardware stack



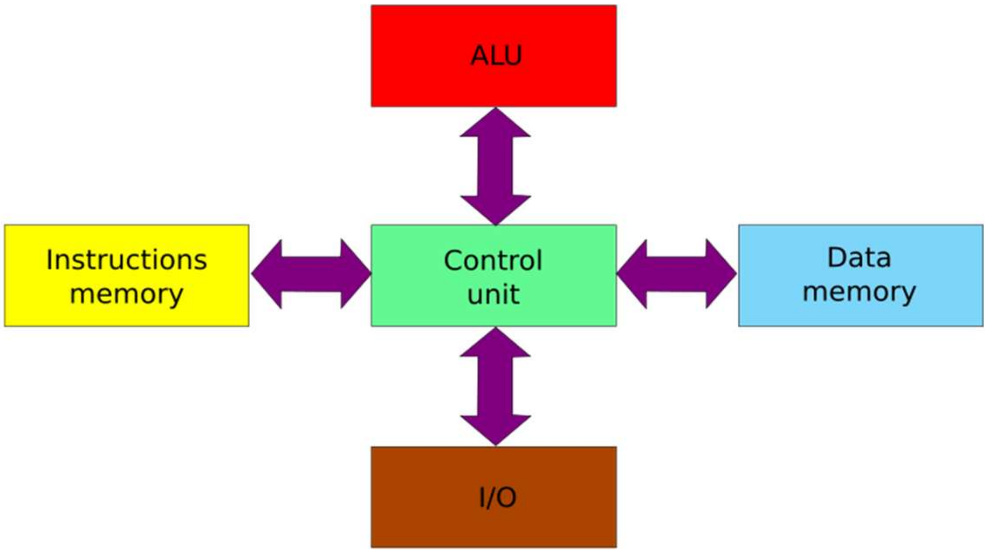
CPU Architectures

- CPU architetcures underpin how data and instructions are stored and processed.
- **Von Neumann**:
  - Data and Instructions stored in same location
  - Stack central to handling multiple tasks/interrupts
- **Harvard**:
  - Separates data an instructions
  - Increased efficiency and security
  - Reduced generality and versatility

CPU Architectures



Von Neumann Architecture



Harvard Architecture

Interrupts and Polling

- Interrupts:
  - Different processes/devices need CPU attention.
  - Interrupts manage how CPU's handle these signals.
  - Stacks provide basis for storing and recalling state while an INT is handled
- Polling:
  - An alternative based on explicitly checking the state of devices/processes
  - Simple to implement but generally considered wasteful of CPU cycles.

Encoders/Decoders/Mulitplexers

- We have covered fundamental combinatorial circuits for data manipulation and transfer
- **Encoders** convert an active input signal into a coded output signal
- **Decoders** selects a single output line based on a coded output
- **Multiplexers** (many-to-one) choose which line to channel data from
- **De-Multiplexers** (one-to-many) choose which output line to channel data to

Signed Number Representation

- Signed numbers can be represented in different ways:
  - Sign magnitude
  - 2's complement
- We can extend either to larger register sizes using sign extension

Sign Magnitude

- Use the most significant bit to represent the sign:
  - 0 is positive
  - 1 is negative
  - E.g., sign magnitude in 8 bits (Big Endian):

0	0	0	0	0	1	1	0	6_{10}
1	0	0	0	0	1	1	0	-6_{10}

2'S COMPLEMENT REPRESENTATION

Find the 2's complement representation of **-6_{10}**

Step1: find binary representation in 8 bits

$$6_{10} = 00000110_2$$

Step 2: ***Complement*** *the entire positive number, and then* ***add one***

00000110 (complemented) → 11111001

(add one) → + 1

11111010

So: $-6_{10} = 11111010_2$ *(in 2's complement form)*

Sign Extension (8-bit to 16-bit)

- e.g. -16 (stored as 2's compliment):

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	1	1	1	0	0	0	0

becomes:

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0

+16

0	0	0	1	0	0	0	0

become

0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

Real Numbers

- Real numbers pose a specific challenge for representing in binary
- Fixed-point representations offer simplicity, but can be wasteful
- Floating point representations standard in modern computers
  – IEEE 754 standard
  – Allows trade-offs of range and precision
  – Requires dedicated FP arithmetic hardware:
    - FPU – floating point unit

Fixed Point Example

Using the fixed<16,7> binary point representation show below, represent the number **25.6640625**:

8	7	6	5	4	3	2	1	0 .	-1	-2	-3	-4	-5	-6	-7

1. Don’t panic. Start by converting 25 to binary:

   25 = 16+8+1 = **000011001.**

2. Then the “decimal” point

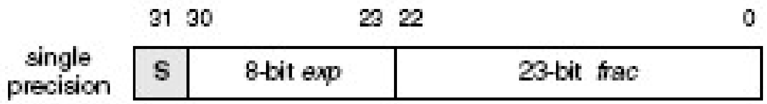
3. Convert .6640625 to binary (starting with 0.5, 0.25, 0.125,...);

   0.5+0.125+0.03125+0.0078125 converts to **.1010101**;

4. Concatenate the two numbers: 000011001.1010101

Floating Point Example

Using the IEEE 754 floating point standard (shown below), represent the number **-273.5** as a 32-bit single precision floating



bit 23 = 2^9=256

binary point between 2^0 and $2^{0.5}$

1. This is –ve, so bit 31 will be 1
2. Convert 273.5 to binary: 256+16+1+0.5 = 100010001.100...(trailing 0s)
3. Shift binary point left to right of bit 23 (count the shifts) (=8) (1.00010001100..)
4. Pad with 0s (LSB) and remove bit 23 (always 1) (.00010001100000000000000)

   the mantissa

5. Add 2^7-1 to the number of shifts (in binary)

   8 00001000

   +127 <u>01111111</u>

   =135 10000111 ← the exponent

6. Concatenate sign bit, exponent and mantissa:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Binary-Coded Decimal (BCD) Example

Using BCD and the fixed point representation show below, represent the number 29.95:

7	6	5	4	3	2	1	0 .	-1	-2	-3	-4	-5	-6	-7	-8

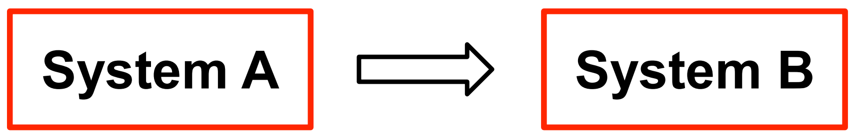
1. It's a 4-digit number so we will need 4 nibbles
2. 2 converts to 0010;
3. 9 converts to 1001;
4. Then the "decimal" point **.**
5. Then 9 (1001)
6. Then 5 (0101)

7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
0	0	1	0	1	0	0	1	1	0	0	1	0	1	0	1

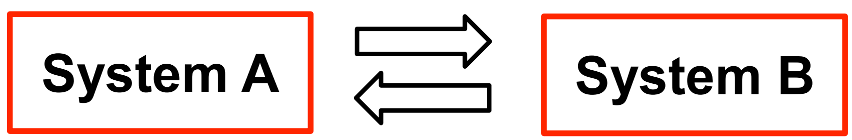
Data Communications Modes

There are three modes of data communication

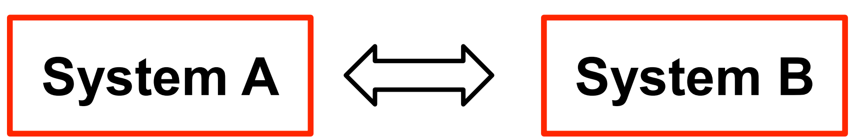
- **Simplex:** Data travels in one direction only



- **Half-duplex:** Data travel in one direction and then other direction, but not the same time

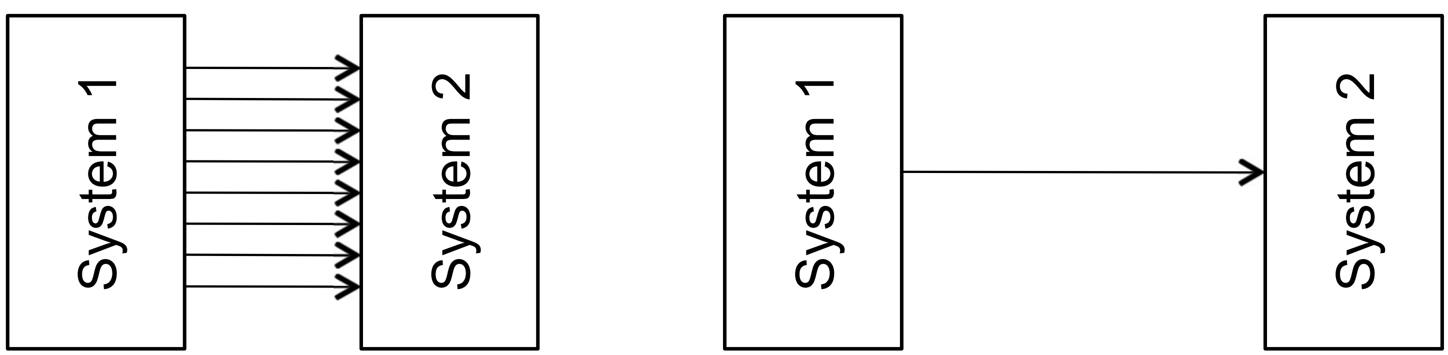


- **Full-duplex:** Data can travel in both directions at the same time



Data Communications

- Almost all computer systems need to send or receive data from other systems or peripherals
- Such data communications can be sent over a number of lines in parallel or over a single line as a serial packet



Data Communications

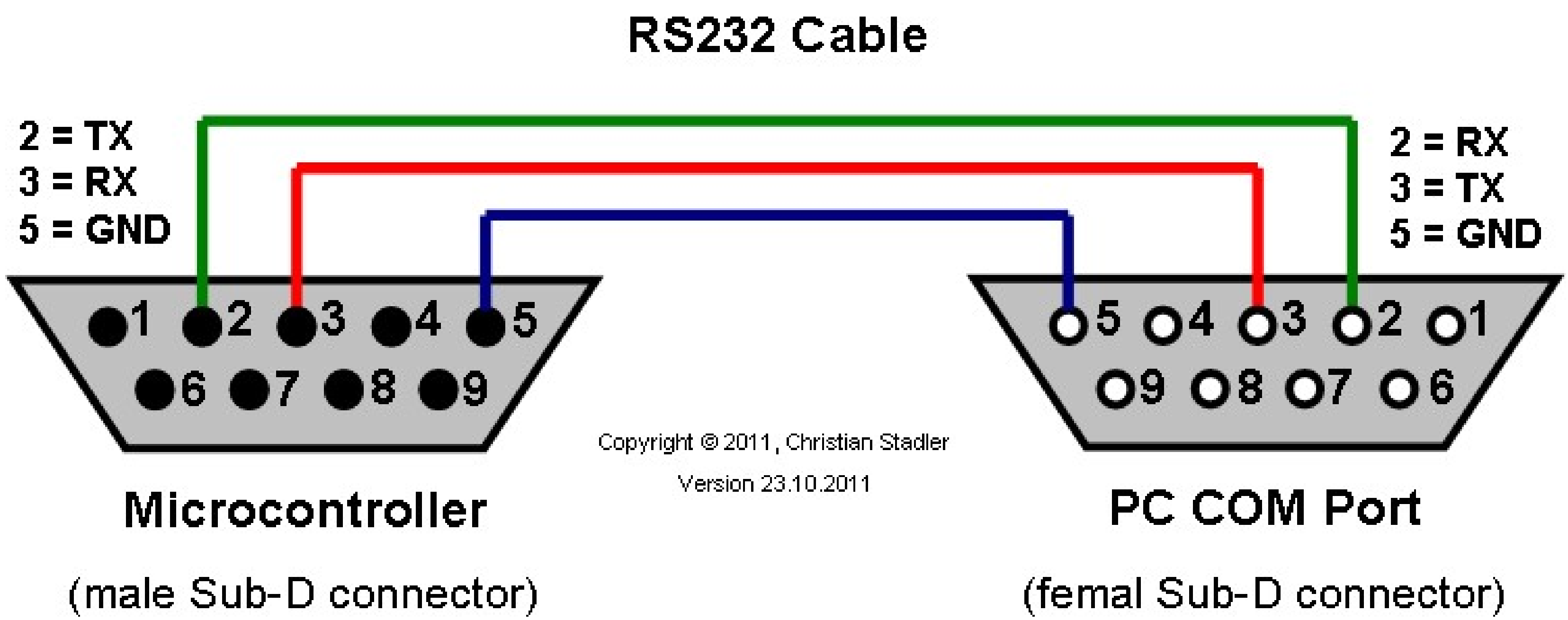
Parallel

- Fast, whole data word transmitted at once
- Only good over short distances: expensive and bulky cables; cross-interference;

Serial

- Slower, all data word transmitted one bit at a time
- Good over short and long distances; less cross-interference; cheaper cables

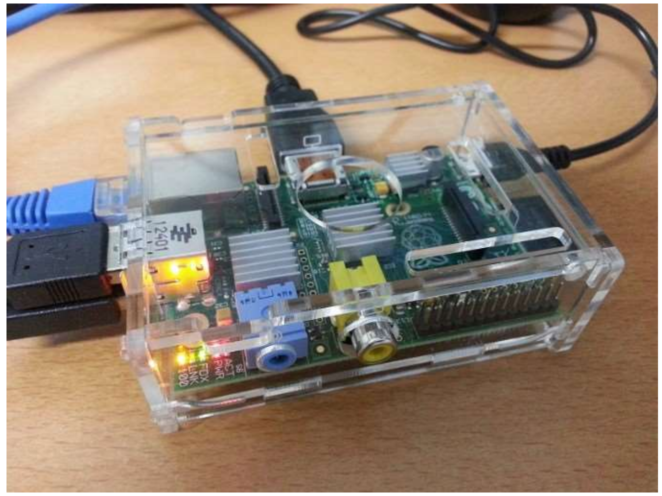
RS-232 Serial Communications



Serial Communications: Parity

- Sending data is always subject to electrical interference or noise that can cause a bit to be misread.
- Parity is a simple means to identify such single-bit errors. The Parity bit is the last bit sent (before the Stop bit)
  - Parity can be 'even' or 'odd'.
  - Even parity means that:

    "the number of logic 1 bits sent (not including the Start bit *but* including the parity bit) must be an even number".
  - The parity bit is therefore set to logic 1 or 0 to ensure the data sent has even parity.
  - The receiver of the data can check the parity of the data received and if it is not as previously agreed (i.e., even or odd) then an error has occurred and the data can be requested to be sent again.



ARM Assembly

Bare-metal programming for Raspberry Pi

RISC versus CISC

- Processor instruction sets can be classified as CISC or RISC

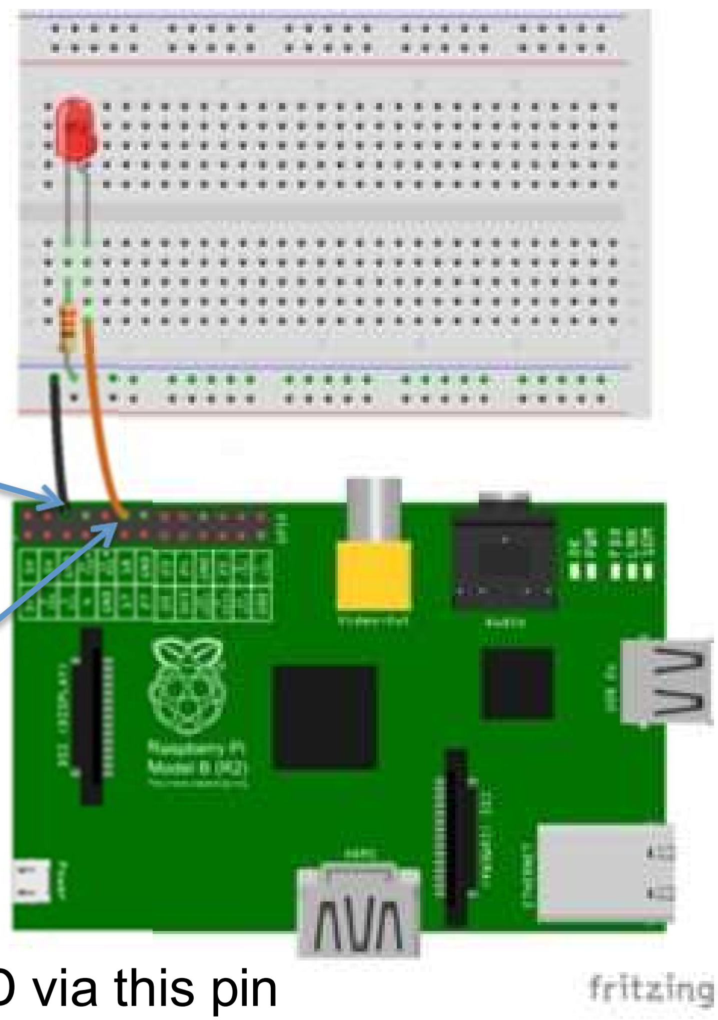
CISC - Complex Instruction Set	RISC - Reduced Instruction Set
Emphasis on hardware	Emphasis on software
Includes multi-clock complex (i.e. specialised instructions)	Single-clock, reduced instruction only
Small code size	Large code sizes
Instructions have variable cycle time	Typical take one cycle
More transistors typically used for storing complex instructions	Less transistors, typically used more for memory registers
Higher power usage	Lower power usage
Typically well suited to multimedia applications	Typically well suited to low powered contexts (eg., mobile phones)

GPIO

- The General Purpose Input/Output chip
- The GPIO chip has 54 registers which can be read, set high or set low.
- They are referred to as GPIO0, GPIO1 …
- Some are connected to physical pins on the RPi board
- Others are connected to hardware on the board.

Wiring it UP

This is the ground pin



We “write” bits to the LED via this pin

See https://www.youtube.com/watch?v=Rd9kvVs1ISQ for my tutorial on wiring this circuit

Turning on an LED

```
BASE = $3F000000   ;  $ means HEX
GPIO_OFFSET=$200000

mov r0,BASE
orr r0,GPIO_OFFSET        ;r0 now equals 0xFE200000

mov r1,#1
lsl r1,#24                ;write 1 into r1, lsl 24 times to move the 1 to bit 24
str r1,[r0,#4]            ;write it into 5th (16/4+1)block of function register

mov r1,#1
lsl r1,#18                ;write 1 into r1, lsl 18 times to move the 1 to bit 18
str r1,[r0,#28]           ;write it into first block of pull-up register

loop$:
b loop$                   ;loop forever
```

Instructions to execute.

Here you can see the following instructions being used:
Mov, orr, lsl, str, b

We'll look at them in more detail shortly

If tests: CMP

- Called 'Compare' in ARM asm
  - Subtracts 2nd value from first, and sets flags accordingly.
  - Loads the **Application Program Status Register (APSR)** with the results of the comparison (done by the ALU).
  - The APSR flags include:
    - N ALU result was Negative.
    - Z ALU result was Zero.
    - C ALU set the Carry bit.
    - V ALU result caused overflow.
  - This register can then be inspected by branch commands
    - We'll come to this !

Programming for input

- To program output, write 001 to address in the program register.
- To program input, write 000 to address in the program register.
  - If we just ***lsl*** a 0 we set all of the other bits to 0.
  - Breaks code for other GPIOs (input and output). Need to be a bit smarter.
- Instead, we can clear a specific bit in a register using **`bic`**

Bit Clear

READING GPIO10

function
select

210 //bit order
000 = input
001 = output
010 = Alt F0
011 = ALT F1
100 = Alt F2
101 = ALT F3
110 = Alt F4
111 = ALT F5

```
BASE = $3F000000 ; Use $FE000000 for 4
GPIO_OFFSET = $200000
mov r0,BASE
orr r0,GPIO_OFFSET ;Base address of GPIO
;read the relevant function register
ldr r1,[r0,#4] ;read function register for GPIO 10 - 19
;clear the 3 bits for GPIO10
bic r1,r1,#7  ;bit clear
str r1,[r0,#4]
```

bits 0-2 = GPIO 0	bits 3-5 = GPIO 1	bits 6-8 = GPIO 2
bits 9-11 = GPIO 3	bits 12-14 = GPIO 4	
bits 15-17 = GPIO 5	bits 18-20 = GPIO 6	bits 21-23 = GPIO 7
bits 24-26 = GPIO 8	bits 27-29 = GPIO 9	
bits 0-2 = GPIO 10	bits 3-5 = GPIO 11	bits 6-8 = GPIO 12
bits 9-11 = GPIO 13	bits 12-14 = GPIO 14	
bits 15-17 = GPIO 15	bits 18-20 = GPIO 16	bits 21-23 = GPIO 17
bits 24-26 = GPIO 18	bits 27-29 = GPIO 19	
bits 0-2 = GPIO 20	bits 3-5 = GPIO 21	bits 6-8 = GPIO 22

A dumb timer

- Variables:
  - r0 = GPIO base address
  - r1 = working memory (for setting bits, registers)
  - use r2 for timing
- mov r2,$3F0000
  loop1:
  sub r2,#1
  cmp r2,#0
  bne loop1

Does a 'busy wait' - uses 100% of CPU

A better timer

- The RPi timer registers:

Byte offset (from BASE)	Size / Bytes	Name	Description	Read or Write
0x3000	4	Control / Status	Register used to control and clear timer channel comparator matches.	RW
0x3004	8	Counter	A counter that increments at 1MHz.	R
0x300C	4	Compare 0	0th Comparison register.	RW
0x3010	4	Compare 1	1st Comparison register.	RW
0x3014	4	Compare 2	2nd Comparison register.	RW
0x3018	4	Compare 3	3rd Comparison register.	RW

WITH CODE (4B)

```
BASE = $FE000000
TIMER_OFFSET = $3000
mov r3,BASE
orr r3,TIMER_OFFSET ;sto
mov r4,$80000  ;stor   delay (r4)
ldrd r6,r7,[r3,#4]
mov r5,r6 ;mov start_time (r5)(=cu
timerloop:
    ldrd r6,r7,[r3,#4] ;read current_
    sub r8,r6,r5  ;elapsed_time (r8)= current_time (r6) – start_time (r5)
    cmp  r8,r4 ;compare elapsed_time (r8), delay (r4)
    bls timerloop ;loop if LE (remaining_time <= delay)
```

- r3=BASE + TIMER_OFFSET + 4 (0x3F003004)
- [r3,#4] means value at ((address in r3) +4)

can't re-use loop label – each must be unique, i.e., loop1, loop2, loop3, …

MANAGING NUMBERS WITH MOV

- The ***mov*** op code is really fast – 1 clock cycle
- It combines the operation (mov) and the operand in the one 32-bit word.
- The ALU/CPU can process this *immediately*
  - No copying from memory (using pointers)
  - No construction of instruction for the ALU.
- BUT:
  - ***It only accepts some numbers*** (those with at least 24 bits set to 0).

MANAGING NUMBERS WITH MOV

- We can break any number up into 1-byte chunks using a bit mask
- We can `AND` the value with 0xFF) and then `ORR` (bitwise add) them with the register.
- e.g.,

```
mov r0,SOME_VALUE ;won't work … but …

mov r0,SOME_VALUE and $000000FF ;copy across
orr r0,SOME_VALUE and $0000FF00 ;1 byte at a time
orr r0,SOME_VALUE and $00FF0000 ;compiles on
orr r0,SOME_VALUE and $FF000000 ;anything
```

Functions in ASM

- Not 'native' to assembly
  - We need to do a lot of the management ourselves
- Argument passing:
  - How do we pass arguments from one function to another
- Storing and recalling register values
  - each function we call will want to use the same registers (only 13 general purpose registers !)
  - How do we manage this ?
- Managing the program control
  - Jumping from one function to another, and then returning back !

Register Management

- **Application Binary Interface** (ABI) sets standard way of using ARM registers.
  - r0-r3 used for function arguments and return values
  - r4-r12 promised not to be altered by functions
  - **lr** and **sp** used for stack management
  - **pc** is the next instruction – we can use it to exit a function call

Software Stack

- A section of RAM managed by the SP (stack pointer) register.
- A sort of 32-bit (64-bit in ARMv8) wide array which starts (element 0) high in RAM and grows down as values are added to it.
- The stack pointer stores the memory location of the last value added (pushed) to the stack.
- Each push decrements SP by 4 (4 bytes per word).
- A pop operation removes the last value in the stack and increments the SP by 4 (4 bytes per word)

Key registers

- Program counter (pc, also r15):
  - Holds the address of the next instruction to execute
- Link Register (lr, also r14):
  - Holds the address of instruction to return to after a function is complete

Delay Function (better)

```
Delay:   ;params: r0 = BASE, r1 = $800000
mov r3,r0
orr r3,$00003000
mov r4,r1    ;~0.5s
ldrd r6,r7,[r3,#4]
mov r5,r6
loopt1:   ;label still has to be different from all the others
  ldrd r6,r7,[r3,#4]
  sub r8,r6,r5
  cmp  r8,r4
  bls loopt1 ;branch if lower or same (<=)
bx lr  ;return to lr - no need to update pc ourselves
```

This way works best with the FASMARM compiler

Factorial

factorialj.asm

- Factorial(n) – n*n-1*n-2*n-3*…*1
- e.g., 4! = 4*3*2*1

```
FACTORIAL:
  sub r1,r1,#1    ;3. r1 approaches 1
  cmp r1,#1       ;4. exit if 1
  beq EXIT
  mul r0,r0,r1    ;total=total*param
  push {r1,lr}    ;2. push onto the stack,
                  ;preserving the PC.
  bl FACTORIAL    ;1. call FACTORIAL
  EXIT:
  pop {r1,lr}     ;pop off the stack
  bx lr           ;RETURN
```

Arrays in ASM

- In ASM we can create arrays, but need to do a bit more of the work.
- Think about what an array requires:
  - An address in memory for the ***first element***
  - A known ***constant offset*** to the next element (in bytes)
  - An uninterrupted contiguous block of reserved space
    - up to the predetermined ***size of the array*** (in bytes)

Arrays in ASM

- Start with a label:
- Follow with the data type and then list the array elements separated by ,

e.g.,

```
myArray:
.int 1,2,3,4,5,6,7,8
myName:
.ascii "James Hamlyn-Harris\0"

myNum:
dw $F0002000
```

the label becomes the address/pointer to the array

dw = define word (32 bit)
db = define byte (8 bit)

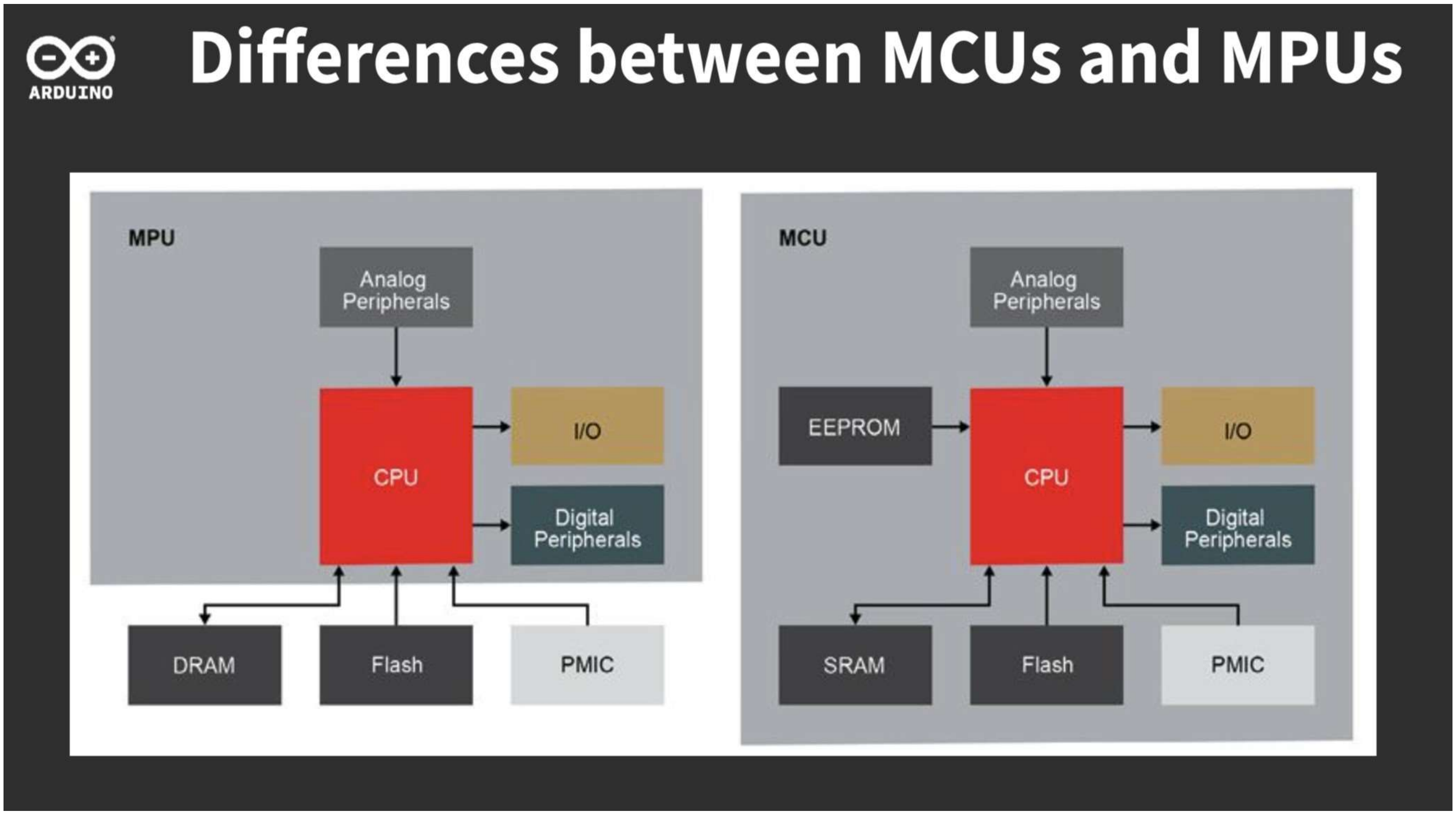
Iterating through an array

- Get the array address and add an offset (index) to it
  - For characters offset is 1 byte
  - For integers offset is 4 bytes
- To get r0 to be set to each value in the array:

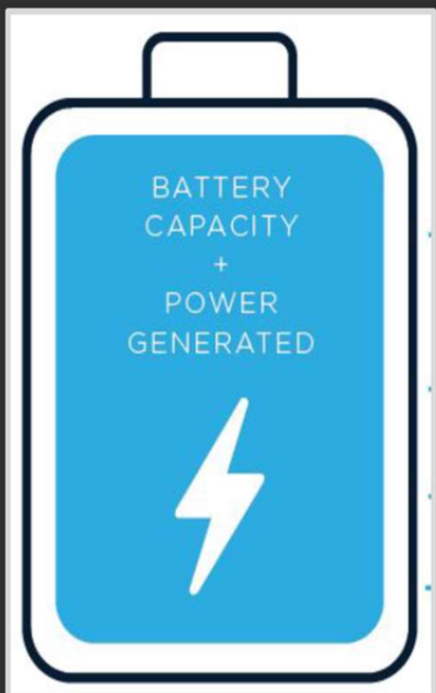
```
; use r5 as the index
; use r4 to point to the array
mov r5,#0 ;i=0
mov r4,myArray ;gets the pointer
loop1:
 ldr r0, [r4,r5]  ; in C: r0 = r4[r5]
 ;do something with r0
 add r5,#4 ;i++
b loop1
```

Multiprocessors and Microcontrollers

- Watch the guest lecture



Multiprocessors and Microcontrollers

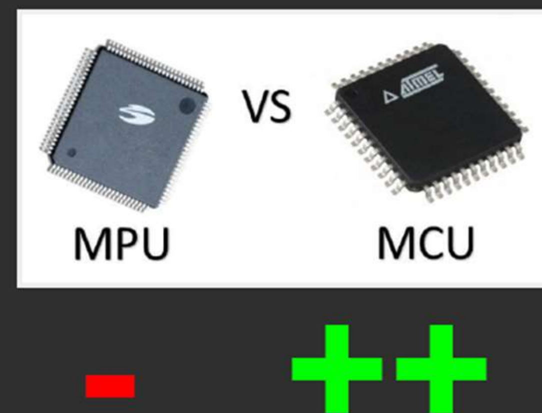


Power management

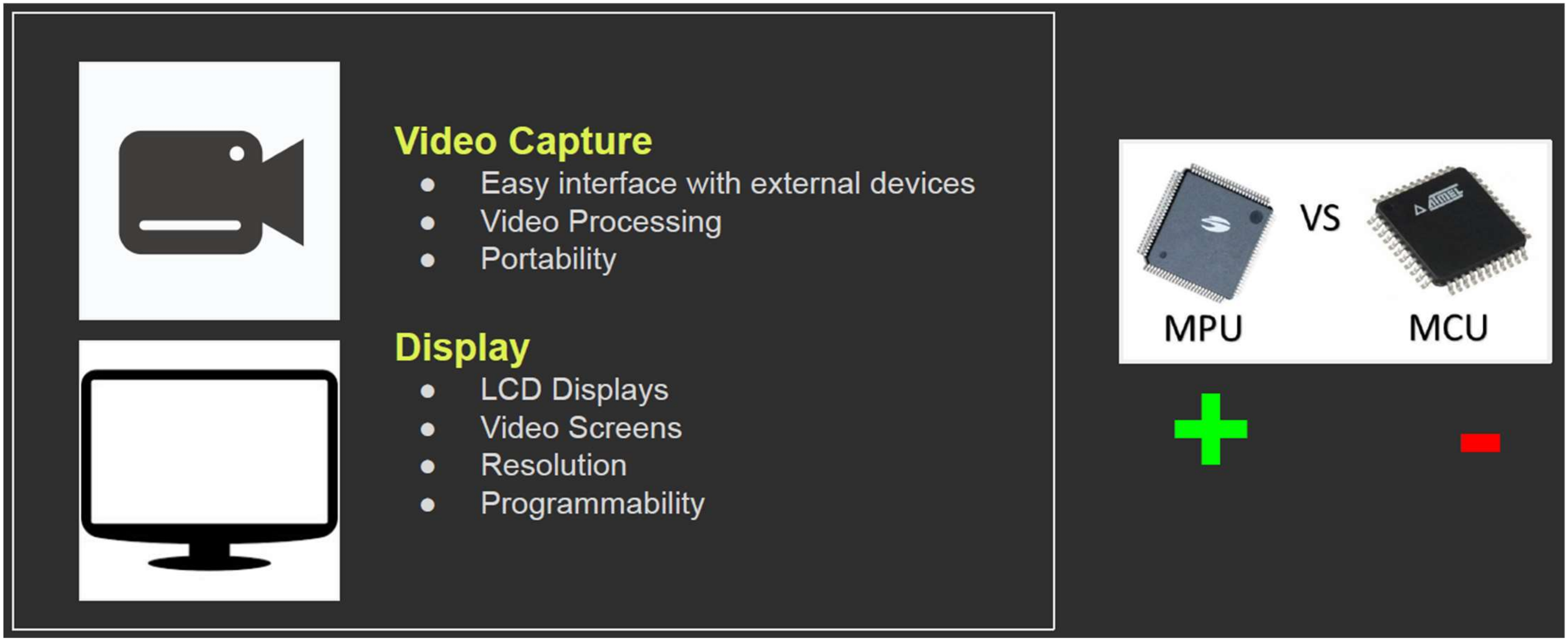
- Sleep Mode
- Wake up on interrupt
- Battery recharge control

Energy harvesting technologies

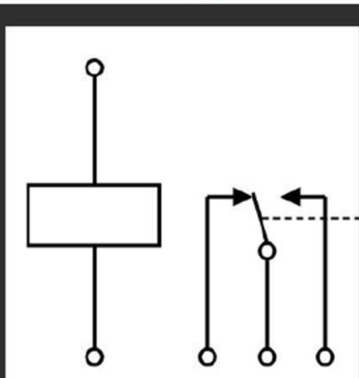
- Solar -> solar panel
- Hydraulic -> micro turbines
- Kinetic -> vibration/rotation harvesters
- Electromagnetic -> EC fields harvesters



Multiprocessors and Microcontrollers



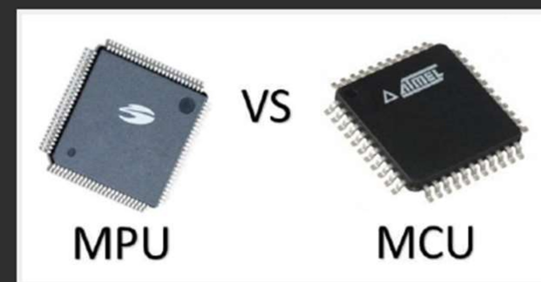
Multiprocessors and Microcontrollers



Interfaces to other devices
- Electrical
- Comm protocol

Wired
- I2C
- RS232, RS485,
- Modbus, CAN-bus, Ethernet

Wireless
- WiFi
- BLE
- SigFox, LoRa,
- GSM, NB



- +

What now?

- Explore!
  - Use the web.
- Use your Raspberry Pi !!
  - Start a project or continue your assignment
  - Muck around with assembly in Raspbian
  - Muck around with GPIO interfacing in Python/C
  - Attach sensors, build a robot/automatic cat feeder

That's it

Thanks for engaging!