Encoding

Represent radix-N numbers

	Binary	Octal	Hexadecimal	Decimal
Conversion Shortcuts	1 bit	3 bits	4 bits	Use a Calculator
Prefix	0b	0o	0x	0d
Examples	101001	51	29	41
	1011	13	B	11

Signed Integer

Sign magnitude

Sign **Magnitude** **Range**

± Value $[-(2^{n-1}-1),(2^{n-1}-1)]$

Sign *Magnitude*

40 = 0 101000 = 0101000
-12 = 1 1100 = 11100
-39 = 1 100111 = 1100111

2's complement

```
- The leftmost bit tells us
  if the number is positive (0) or negative (1).
- To make a negative number,
  flip all the bits of the positive number and add 1.
- To make a positive number,
  substract 1 first,flip all the bits of the negative number.
```

Fixed Point

```
- integer bits determin range
- Fractional bits determine accuracy
- cannot represent everything clearly,like 1/3
```

Decimal	100s	10s	1s	1/10s	1/100s	1/1000s
Binary	4s	2s	1s	1/2s	1/4s	1/8s

Floating Point

```
- Exponent usually in 2's compliment
- Choose exponent to cover range
```

Sign **Exponent** **Mantissa**

$\pm$ 2^x $0.b_0b_1b_2\dots.$

$\pm$ 2^x $1.b_0b_1b_2\dots.$

max error

```
Max error (floating point) = (1/2) * (Max number - Second max number)
```

add

```
To do this we have to shift the numbers so they have the same exponent
and then line them up and add them.
```

$(2^3 * 0.1001_2) + (2^1 * 0.1010_2)$

$= (2^3 * 0.1001_2) + (2^3 * 0.00101_2)$

$= 2^3 * (0.1001_2 + 0.00101_2) = 2^3 * 0.10111$

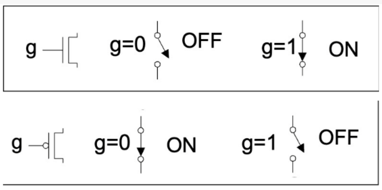
$\approx 2^3 * 0.1100_2$ (can only keep 4 mantissa bits so we round to nearest)

```
We often have an incorrect answer when adding in floating point
because two numbers may be very different sizes and so,
when you change their exponents to be the same size, there are not enough
mantissa bits to adequately represent the number.
```

Transistors, Boolean Algebra and Digital Logic

Transistors

```
MOS transistors are switches
```



Boolean Algebra and Digital Logic

Combinational Digital Logic

XOR (异或门)

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR (同或门或异或非门)

A	B	A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1

Gate	Symbol	Operator
AND		$A \cdot B$
OR		$A + B$
NOT		$\overline{A}$
NAND		$\overline{A \cdot B}$
NOR		$\overline{A + B}$
XOR		$A \oplus B$
XNOR		$\overline{A \oplus B}$

SOP/POS

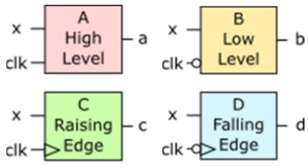
```
- SOP(Sum of product) : R = A·B + C·D + E·F
  (Find lines which outputs are 1)
- POS(Product of Sum) : R = (A + B)·(C + D)·(E + F)
  (Find lines which outputs are 0)
```

Sequential Digital Logic

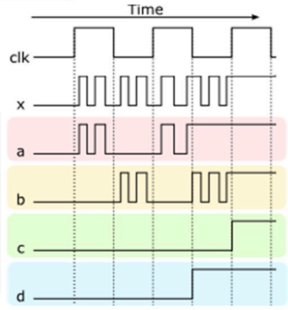
Clock, latches and flip-flops

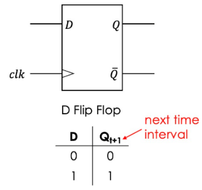
```
Clock
+ Copy input when clock says so
+ Otherwise don't change value
```

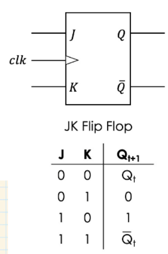
```
Latches
+ Level-sensitive: Changes or preserves state based on the
  level (high or low) of a control signal.
+ Transparent when enabled: When the enable signal is active
  the output immediately reflects changes in input.
```

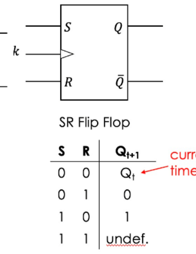


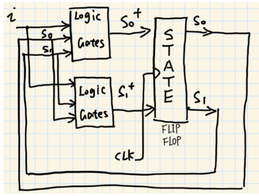

```
Flip-Flops
+ Edge-sensitive: Responds and changes state only at
  a specific edge (rising or falling) of the clock signal.
+ State preservation between clock edges:
  Maintains its state stable in the intervals between clock signal edges.
```











Finite State Machine

```
1. Write the truth table from the inputs and state bits, to future state bits.
2. Implement each of the outputs with a circuit.
3. Connect the outputs of the function to flip flops.
4. Connect the output of the latch to the inputs of the circuit.
```

Memory

Types of memory/Visualization

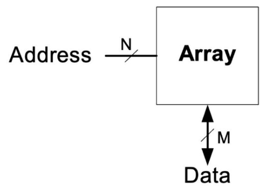
Register

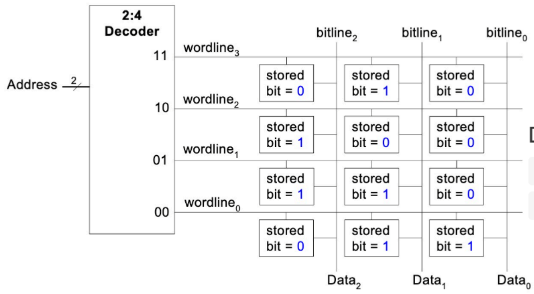
A register is just a collection of **D-Flip-flops**

It allows you to represent 'words' (your encodings e.g. 4-bit unsigned binary)

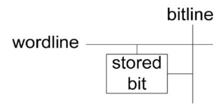
RAM(ROM)

```
- KB 2^10 Bytes
- MB 2^20 Bytes
- GB 2^30 Bytes
See graph below
- N: Amount of the element
    if Bits is given,actual number of address is 2^N
- M: bits of 'word'
```



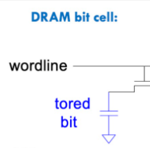


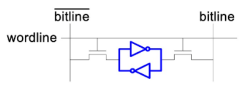
Bit cell



DRAM bit cell vs SRAM bit cell

```
DRAM : samller but slower
SRAM : bigger but faster
```

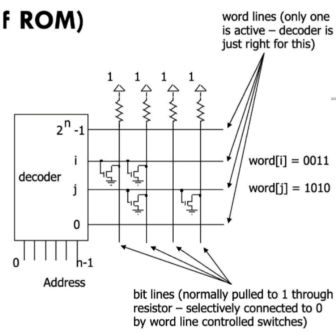




How does ROM work

```
Falsh(闪存) is a variant of ROM
```

f ROM)



Store data in Memory

Read/Write

```
Follow Little/Big-Endian
```

Big-Endian Store 0x1234 **Little-Endian** Store 0x1234

```
Address N:   0x12   Address N:   0x34
Address N+1: 0x34   Address N+1: 0x12
```

Store Array in Memory

Memory Allocation

```
- Each element in the array uses memory according to its data type.
    For example, an int might use 4 bytes (depending on the platform)
    while a char typically uses 1 byte.
- The total memory used by the array is
    the size of the data type multiplied by the number of elements.
```

Accessing Elements

```
base_address + (element_size * index)
```

Data Types

Boolean	1 bit
Character	2 bytes
Integer	4 bytes
Long	8 bytes
Float	4 bytes
Double	8 bytes

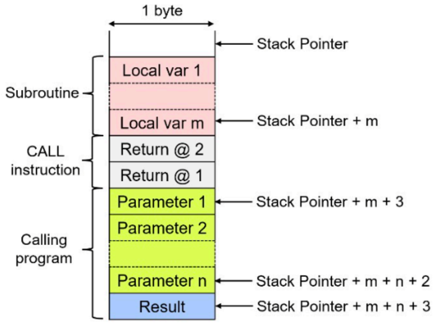
Load

```
LDI Rd,K
; Load Immediate
; R16-31,8 bits number
LD Rd,X/X+/-X
; Load Indirect from Data Space
to Register using Index X/Y/Z
LDD Rd,Y/Y+3/Y-1 ;
only support Y and Z
LDS(32-bit)
LDS Rd,k ; Load Direct from Data Space
; 0 <= k <= 65535
; PC = PC + 2
; 32 bits opcode
```

Store

```
ST X/-X/X+,Rr
; Store Indirect From Register
to Data Space (X/Y/Z)
STD Y/Z + q
; Store Indirect From Register
to Data Space with Displacement (only Y/Z)
; q is an integer in [0,63],6 bits
STS k,Rr  ; this is a 32-bit version
; Store Direct to Data Space
; 0 <= k <= 65535
; PC = PC + 2
; 32 bits opcode
!!! there is a 16 bits version
```

1 byte



Stack

Avr

```
.section .data
;define variables

.section .text
;doing calculations
.global asm_function

asm_function:
;here is your main function
ret

.end
```

While Loop

```
loop_comparison: ;

start_loop_body:
[loop body code here] ;
JMP loop_comparison

end_loop:
[code here] ;
```

For loop

```
init_loop_iterator:
[initialise_loop_iterator]

loop_comparison :
CP R1, R2 ;
BRLO end_loop;

start_loop_body:
[loop body code here] ;
[modify_loop_iterator] ;
JMP loop_comparison  ;

end_loop:
[code here] ;
```

IF/ELSE

```
[Before-if/else code here
cp r1, r2 ;
br.. else ;

if:
[IF code here] ; You didn
jmp end_if

else:
[ELSE code here] ; You di

end_if:
[code here] ;
```

```
JMP k
; 0 <= k < 4M
; PC = k
```

Branches

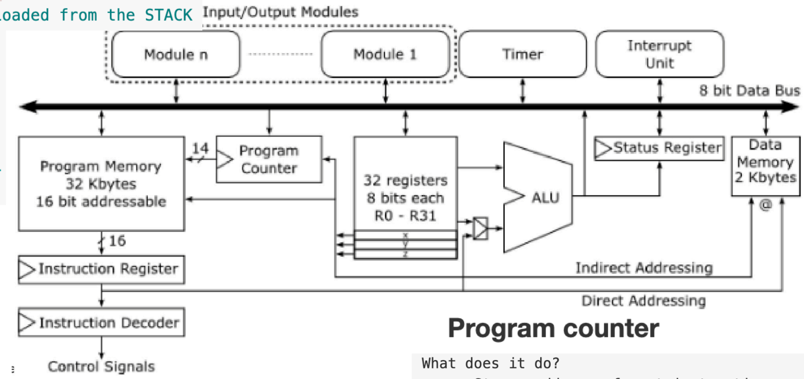
```
CP Rd,Rr
; Rd >= Rr -> Flag C = 0
; Rd < Rr -> Flag C = 1
; Rd = Rr -> Flag Z = 0

CPI Rd,k
; Rd >= k -> Flag C = 0
; Rd < k -> Flag C = 1
; Rd = k -> Flag Z = 0

CALL k
; Stack = PC + 2
;SP = SP - 2

RET
;SP = SP + 2
;The return address is
loaded from the STACK

BREQ k
; branch if Flag Z = 1
(Rd = Rr / Rd = k)

BRNE
; branch if Flag Z != 1
 (Rd != Rr / Rd != k)
BRSH k
; branch if Flag C = 0
 (Rd >= Rr / Rd >= k)

BRLO k
; branch if Flag C = 1
(Rd < Rr / Rd < k)
```

Computer Architecture

CISC vs RISC

```
RISC
    - can do everything that CISC can do.
    - instruction faster(in a bisic way,RISC runs faster,
        but CISC can use pipelining to be faster)
    - shorter instructions
    - simpler compiler
CISC
    - takes longer to decode
    - has direct access to operands in memory.(RISC not)
    - almost always has less lines of code.

Why different instructions? Size/Power/Performance/Mistakes
RISC stands for
"Reduced Instruction Set Computer".
This architecture is characterized by:

- Instructions have at most 2 operands.
- There are 32 general-purpose registers
  that arithmetic operations (ADD, SUB, MUL) can operate over.
- The first register in an instruction is both the
  destination and one of the sources (e.g., `ADD R15, R16`).
- There is an instruction (`LOAD`) to load a value from
  a memory location with a symbol name
  (e.g., `LOAD R2, x` to load variable `x` from memory to `R2`).
- There is an instruction (`STORE`) to
  store a value from a register to a memory location.

CISC stands for "Complex Instruction Set Computer".
This architecture is characterized by:

- Instructions have three operands.
- There are 32 general-purpose registers.
- The first operand is the destination, and the second and
third are source operands (e.g., `ADD R2, R15, R16`).
- There is an instruction (`LOAD`) to load a value from a memory
 location with a symbol name (e.g., `LOAD R2, x` to load
 variable `x` from memory to `R2`).
- There is an instruction (`STORE`) to store a value
  from a register to a memory location.
- Arithmetic operations (ADD, SUB, MUL) can
  have any of its source operands in memory.
```

AVR Microprocessor



Program counter

```
What does it do?
    - Stores address of next instruction
    - Increments when move to next instructio
What type of digital circuit is it?
    - Register
    - Adder(conditional)
    - Multiplexer
```

Program memory

```
What does it do?    Store instructions
What operations does it perform?    Read /write
What are it's properties?
    - Each cell is ? bit
    - Read/write is always ? bits
    - Different to data memory
```

ALU

```
What does it do?
    - Calculations
    - Add/Sub/Conjunction/Negation
How many bits can it add?
    - Tie to the registers
```

Register File

```
What is it?
    - Tiny, local memory
        * Temporary storage
What digital circuit is this?
    - 32 8-bit Registers (r0 to r31)
What are X,Y,Z?
    - Design decision
    - Enable 26,27 (x), 28,29 (y),
    30,31 (z) to be 16-bit manipulations
```

Instruction register

```
What does it do?
    - Remembers current instruction
```

Status Register

```
What does it do?
    - Stores flags of special conditions
    - Zero/overflow
What does it listen to?
    - ALU
Where is this information used?
    - Subsequent instruction
```

Data memory

```
What does it do?
    - Stores integers Arrays...
2KB. How many address bits?
    - Needs 11
    - Actually has 12
        * Pretend larger memory
        * Gives 256 additional fake address
        * Allows some faster operations.
```

Instruction decoder

```
What does it do?
    - Open instruction, decide what to do
    - Controls rest of circuit
        * Operation: Adding/subtracting
        * Operands
        * Where to write result
```

```
boolean A, B, C, D, E, F, G;
int count = 0;

void setup() {
  pinMode(2, INPUT);
  pinMode(6, OUTPUT);  // G
  ......
  pinMode(12, OUTPUT); // A
  attachInterrupt(0, x, FALLING); //pin2
  Serial.begin(9600);
}

void loop() {
  Serial.println(digitalRead(2));
    switch (count) {
      case 0: A=1; B=1; C=1; D=1; E=1; F=1; G=0; break;
      ......
    }

    digitalWrite(6, G);
    ........
    digitalWrite(12, A);
  delay(500);
}

void x()
{
  count = (count + 1) % 10;
}
```

```
void forwardMedium() {
  Serial.println("Moving forward at medium speed");
  leftServo.writeMicroseconds(1600);
  rightServo.writeMicroseconds(1400);
  delay(1000);
  stopMotion();
}

void turnLeftSlow() {
  Serial.println("Turning left at slow speed");
  leftServo.writeMicroseconds(1450);
  rightServo.writeMicroseconds(1450);
  delay(1000);
  stopMotion();
}

void turnRightSlow() {
  Serial.println("Turning right at slow speed");
  leftServo.writeMicroseconds(1550);
  rightServo.writeMicroseconds(1550);
  delay(1000);
  stopMotion();
}
```

```
pinMode(leftPhotoResistor, INPUT);
pinMode(rightPhotoResistor, INPUT);
```

```
#include <Servo.h>

Servo leftServo;
Servo rightServo;

const int leftPhotoResistor = A0;
const int rightPhotoResistor = A1;
```