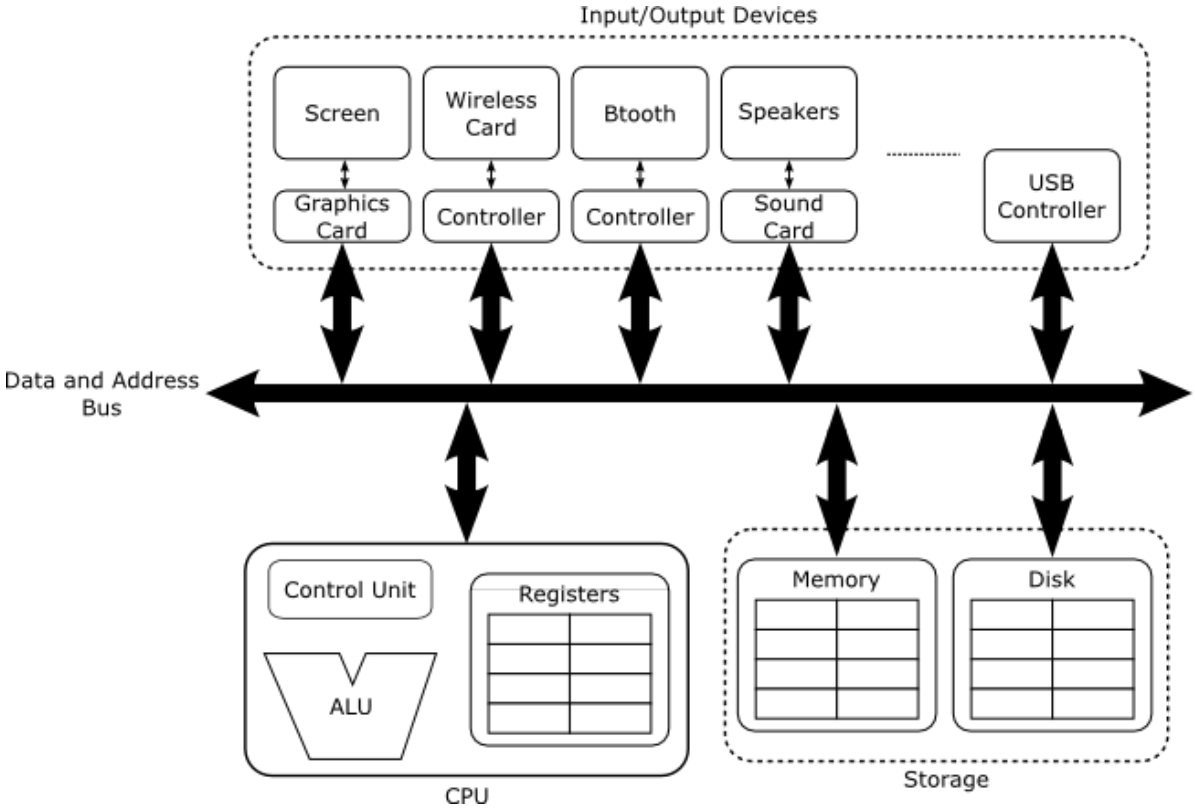
*abstraction** is a simplifying process use to reduce the level of complexity of complex problem.

Structure of a computer system



At least one processor (CPU)

- **registers**: store some temporary data
- **arithmetic-logic unit(ALU)**
- **control unity**: giving the orders to execute the sequence of machine instructions.

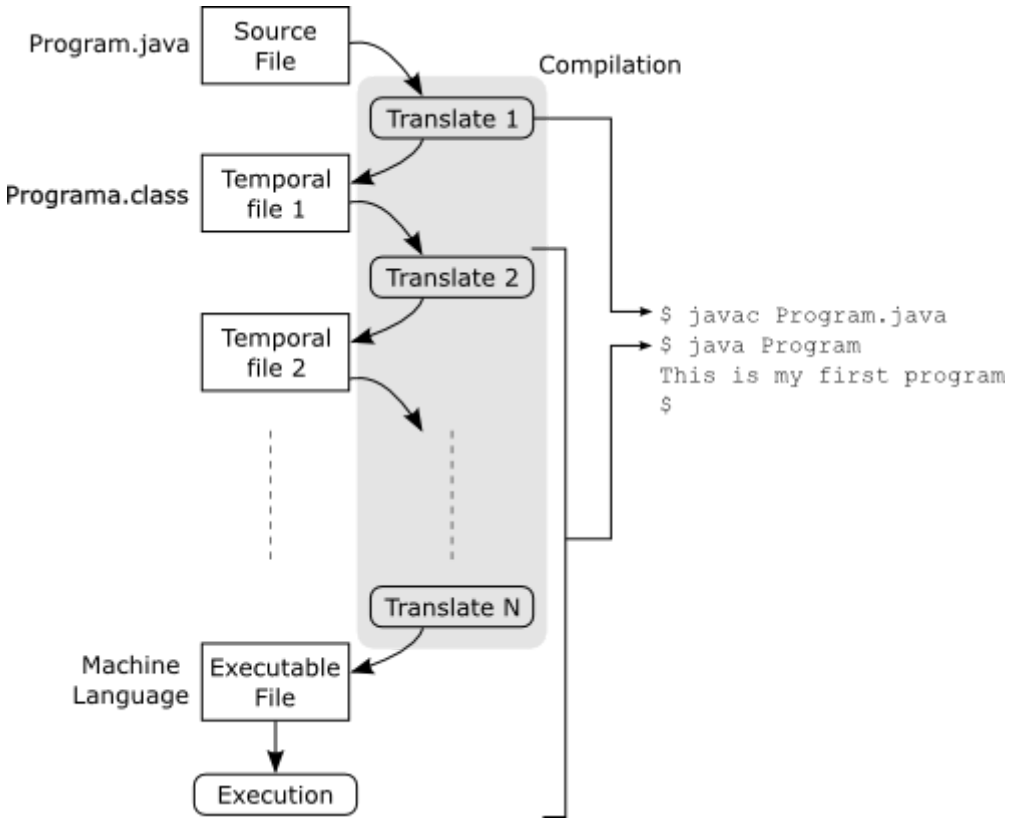
Storage devices

- **static RAM**
- **dynamic RAM**
- **Flash memory**
- **hard drive**
- ...

Input/output devices

- **screen**
- **graphic card**
- **USB controllers**: keyboard, mouse, camera, ...

Definition of a program



- `javac` applies the first step and creates a temporary file with extension `.class`
- `java` executes the program
-

The assembly language: A human readable format to represent the machine language.

All elements need to be encoded with binary logic

$$N \leq 2^n$$
$$n \geq \log_2 N$$

1. With n bits, encoded up to 2^n elements
2. Encode N elements require $\log_2 N$

sign and magnitude use the most significant bits to represent sign, 0 as positive and 1 as negative

- range $[-(2^{n-1}-1), 2^{n-1}-1]$, $2^n - 1$ in total

but it use two bits to represent 0, for example, 000 and 001

2s complement only use 1 bits to represent 0, all zero

- range $[-(2^{n-1}), 2^{n-1}-1]$, 2^n in total

Decimal to Binary

if the number is positive

- write sign and magnitude

if the number is negative

1. Obtain the base 2 encoding
2. replace 1 with 0, replace 0 with 1
3. add 1

Binary to Decimal

if the number is positive

- normal way

if the number is negative (same)

1. Obtain the base 2 encoding
2. replace 1 with 0, replace 0 with 1
3. add 1

Floating point

maximum error for fixed point = find the last significant bit, the bit after that will be the maximum error, for example, 1.111's maximum error is 2^{-4}

maximum error for floating point = write the number into representation system, convert it back to the base 10 number, compare this number with the origin number

mantissa (significand): right hand size

floating point representation = **fractional component(mantissa)** + **exponent of a certain radix(base)**

the weight of decimal part obtained by decreasing negative powers of the base.

In floating point representation, move the floating point until the first significant bit is the first position to the right of the radix point.

Step:

1. turn base 10 to base 2
2. move radix point until the most significant bit of mantissa is 1 (when facing a domain, represent the bigger number)
3. record sign, mantissa and exponent (represent the exponent first, then deduce the mantissa)

mention: some time exponent is also encoded with 2s complement

the sign of mantissa always take 1 bit

base 8 start with 0

Range

For example, in one scheme the 14 bits are allocated as 1 for the sign, remaining 7 for the mantissa and 6 bits for the exponent. If we assume that the exponent is encoded with 2s complement values,

$$[-0.1111111 * 2^{-31}, 0.1111111 * 231]$$
$$Negative : [-0.1111111 * 2^{31}, -0.1 * 2^{-32}]$$
$$Positive : [0.1 * 2^{-32}, 0.111111 * 2^{31}]$$

Accuracy and precision

The precision is affect by the assignment of mantissa and the exponent

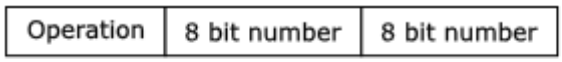
Accuracy: how close the number is to number we want

Precision: the length of mantissa

Overflow and Underflow

The set ual-1

Instruction = Operation + Operand

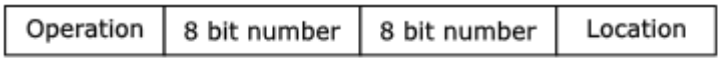


add 0x23 0x34
sub 0x4F 0xBA
mul 0x12 0xFF
div 0xFF 0x08

use 24 bits(18 bits need), 3 bytes, the last 6 bits are always 0, called filled bits

The set ual-2

Instruction = Operation + Operand + Destination

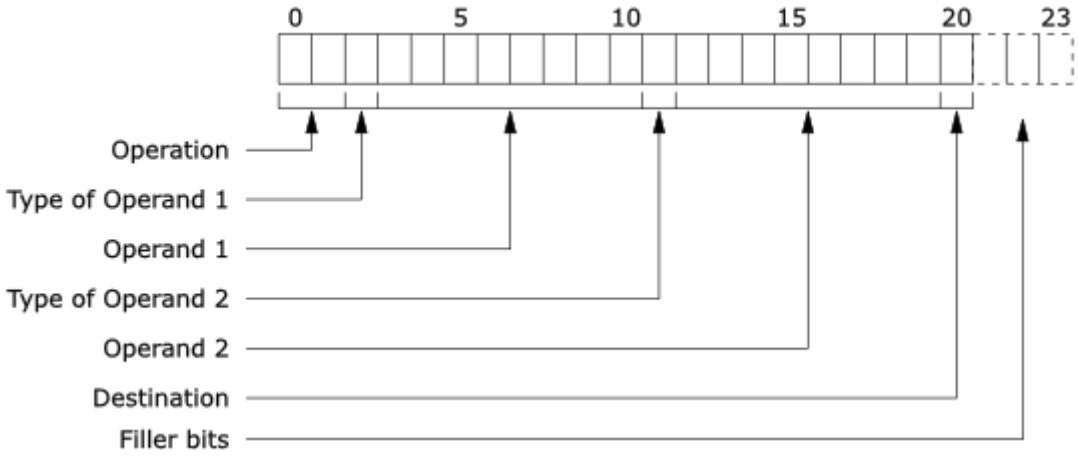


add 0x23 0x34 Location_A
sub 0x4F 0xBA Location_B
mul 0x12 0xFF Location_B
div 0xFF 0x08 Location_A

The forth element use to represent the location to store the result, location only take 1 bit, 1 or 0

use 24 bits(19 bits need)

The set ual-3



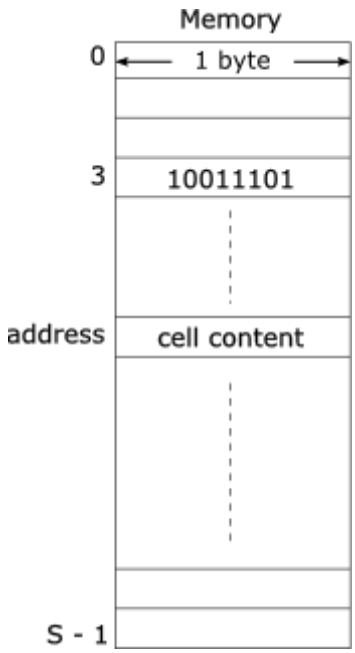
if type of Operand is 0: read follow 8 bits as input value

else ignore follow 7 bits, read the last bit to obtain the value location

fixed format: all instructions have the same length

Variable format: instructions have different size depending on the information they contain, the decoding must be done incrementally

each processor has its own machine language



RAM(random access memory), memory cell size is 1 byte, memory address start counting from 0

made of a set of transistors and gates

volatile, when power is turned of, the data is lost

two types of RAM: static and dynamic

- SRAM: as above, not need to refresh
- DRAM: use capacitors to store value, is charged -> store 1, discharged -> store 0
    - simpler, so higher capacity
    - have to refresh, it leak current
    - refresh all cells continuously before lose charge
    - constantly being read or refreshed

Memory operations

reading: receive an address, returns the content

writing: receive an address and a value, write the value in the cell

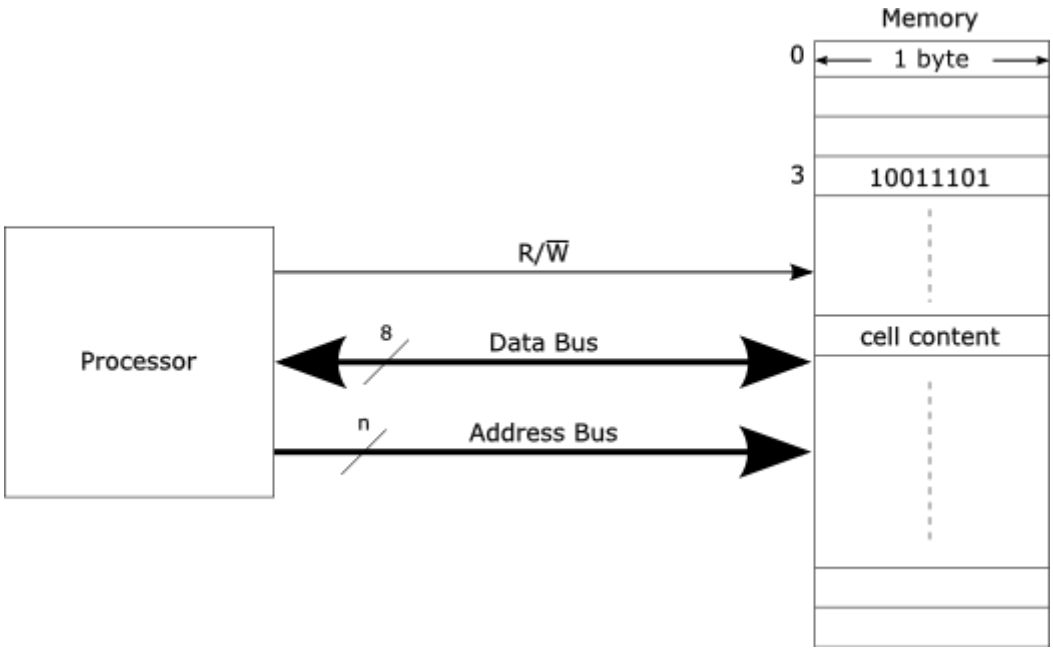
data initially in memory is **underfined** or **garbage**

both data and address need to be encoded in base 2

relationship of size (S) and bits use to encode the addresses $S \leq 2^n$

capacity = number of cell * cell size

Connection between memory and processor



Bus is a set of wires in which several circuits can read and write binary values

Two buses + one signal

- address bus: send addresses of operation to memory chips
- data bus: send data processor -> memory chip while writing, memory chip -> processor while reading
- one bit signal: $r/\bar{w}$ to tell processor the type of operation, 0 -> write, 1 -> read

the slanted line use to denote number of bits in **binary**

multiple byte operations: if a memory is capable of reading and writing data in groups of x bytes, then instruction `Write(address, value)` will write value in the address and x position after that, same rule in `Read`

Data storage

For hight level programming languages, compiler is in charge of mapping the complex data structures to the memory of the computer system. Different type, different size. It will use as many consecutive cells as needed, the address of the first cell that data structure is stored in is called data address

Boolean

1. Store in a single bits, need to know address and the position of the bit inside the byte, need additional operation when extracting or inserting the value from or into the memory cell
2. One booleans in one byte, waste the rest bits, but faster

Characters

Depending on the encoding scheme

ASCII -> 8bits, if size of memory is 8 bits, 1 cell 1 char

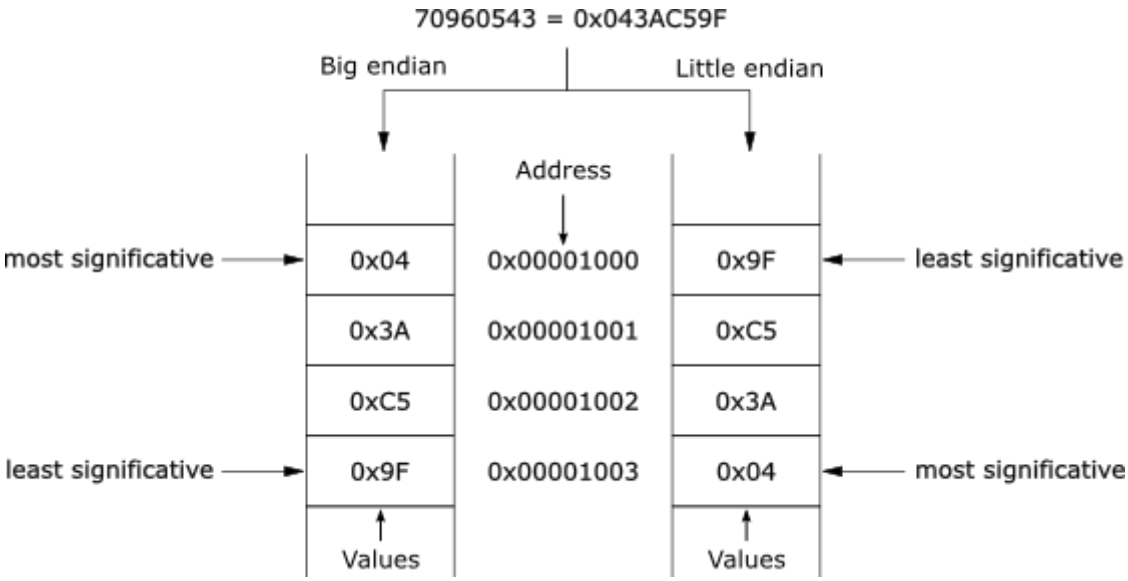
UTF-16, two size 1 char

integers and natural numbers

size: 2, 4, 8, 16 bytes

Can be stored from least significant to most significant, vice versa (equally used)

two way to store 0x043AC59F



little endian: first byte stored is the least significant

big endian: first byte stored is the most significant

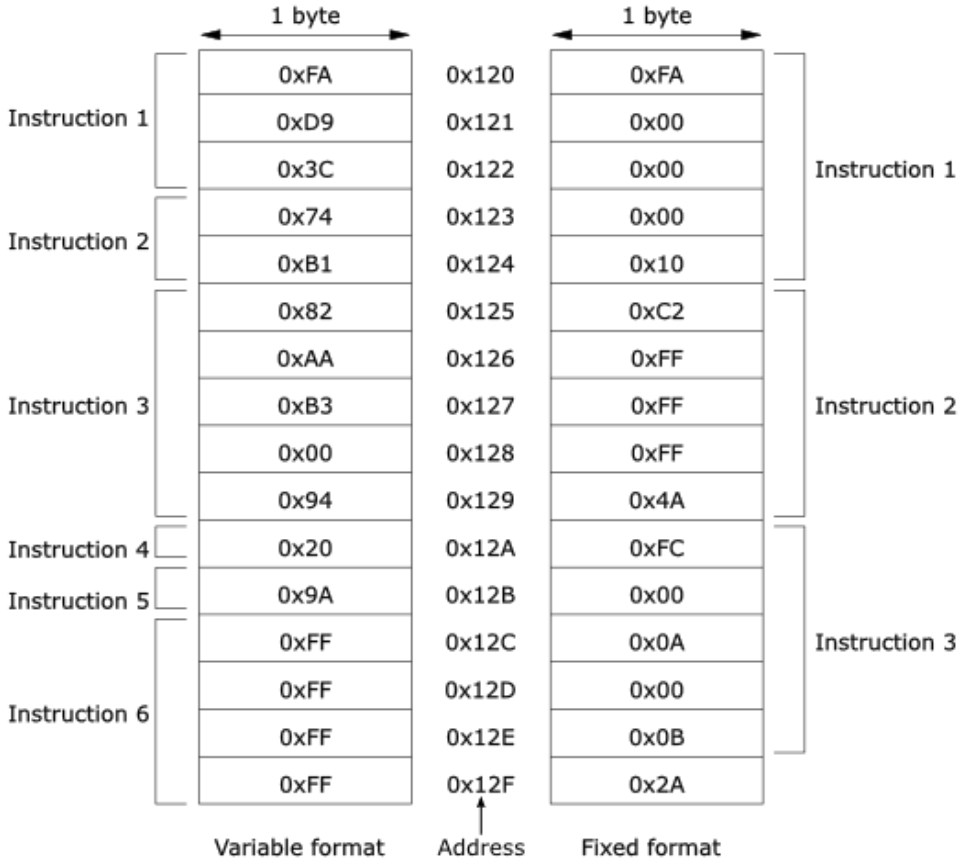
if there are two processor in one machine with different policies, two strategies are used, need operations to swap the order in which the bytes are manipulated.

stack: when little endian, need to extend representation, only need to use extra memory cells at the end

Storage for machine instructions

two types of processors according to length of instruction:

- fixed length for all the instructions: easy to access the next instruction
- variable instruction length: need to deduce the next instruction from current instruction's address and size -> first obtain one instruction from memory -> interprets its content, deduces its size -> add ahead to address and obtain the next one



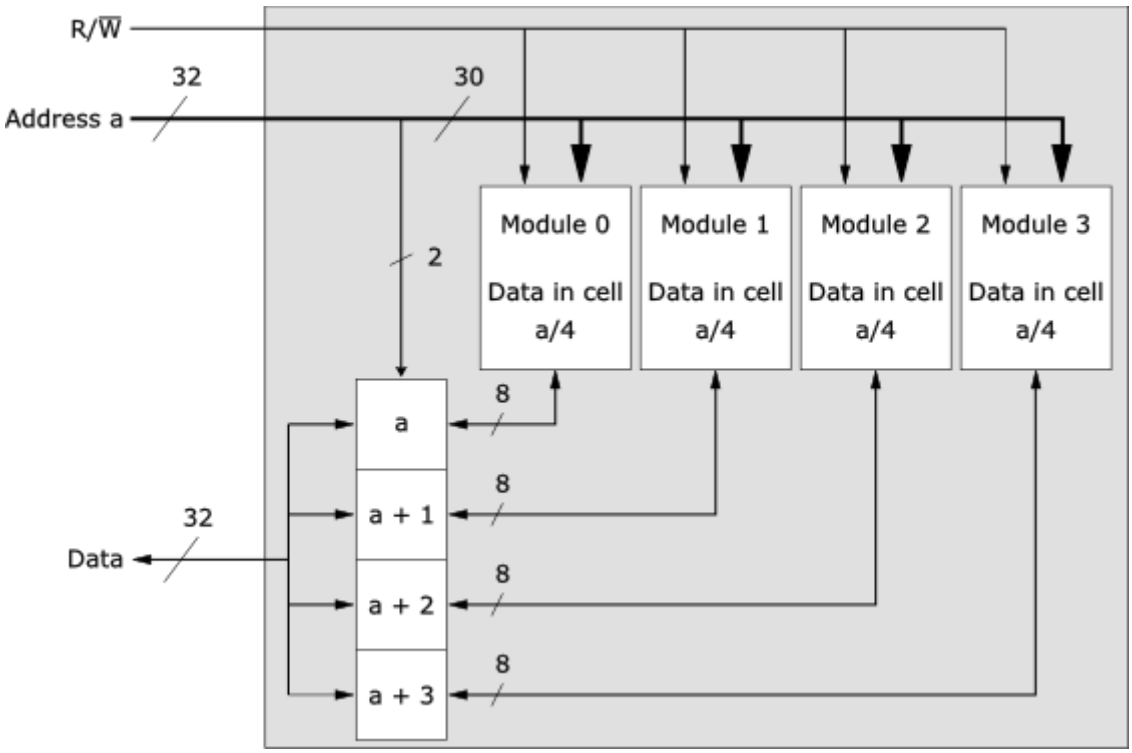
no need to distinct little and big endian

bottleneck: read or write operation is much slower then processor

solution: perform read and write operations for several cells at the same time, some system allow to operate blocks of consecutive memory cells

- combining multiple memory chips or modules to behave as a regular memory, operate several bytes simultaneously

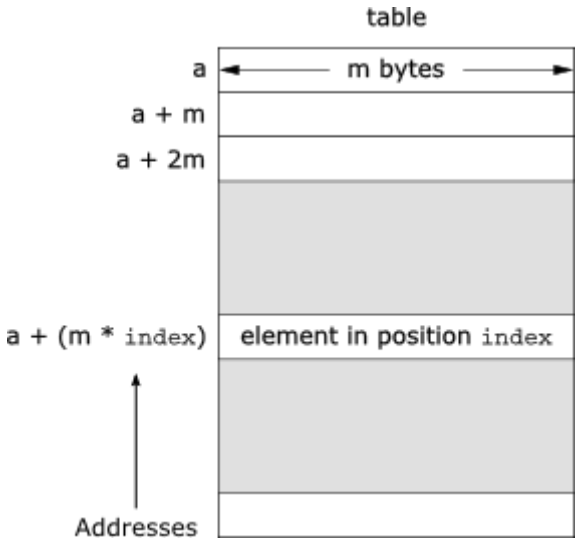
- connect memory chips in parallel, use position%(size of combination) to allocate cells into memory, so consecutive cells are stored in different modules
- require blocks of these cells aligned with memory addresses that are multiple of size of blocks(4 in a group, aligned multiple of 4, can return 0, 1, 2, 3, can't return 1, 2, 3, 4)
- obtain 30 bits of 32 bits of address, because the first module contain cell end of 00, second 01, third 10, fourth 11, not need to obtain the last 2
- each module provides one of the byte in the group
- if want to obtain data which is not aligned, need to read or write more times



Storing an Array

If the elements starting from address *a* and each occupies *m* bytes

$$address(table[index]) = a + (m * index)$$



two way to locate the end of the array:

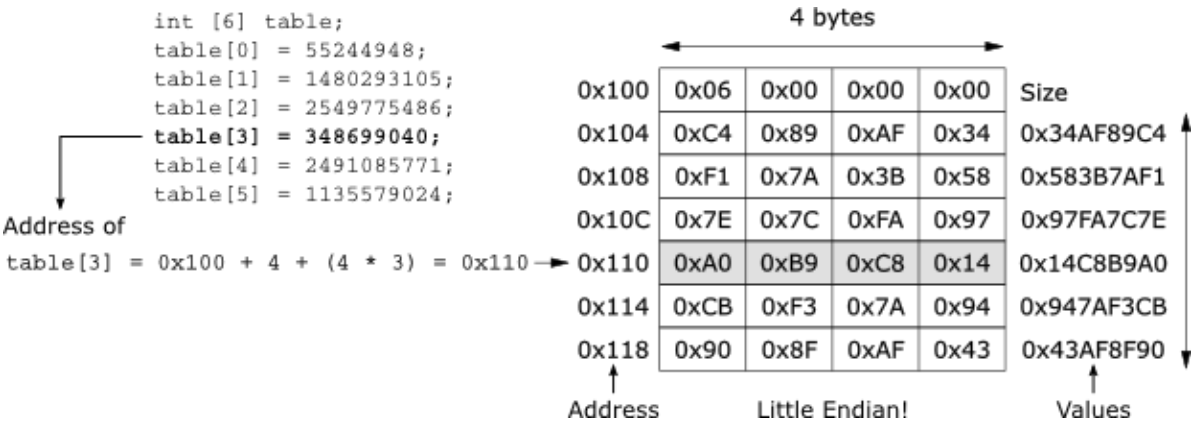
- placing a special value (no useful value) to mark the end of the table, for example, '\0' to mark the end of String
- for integers or naturals, storing the size of the array in another memory location, when traversed it, read this memory

stored in **Java**

If table with n elements is defined in Java, index must satisfies $0 \leq i < n$

Check the value while the program is running and right before the access is performed, is false, ArrayIndexOutOfBounds

store the size of table in the first memory positions of the table, so when calculate the size of an array, add **4 bytes**

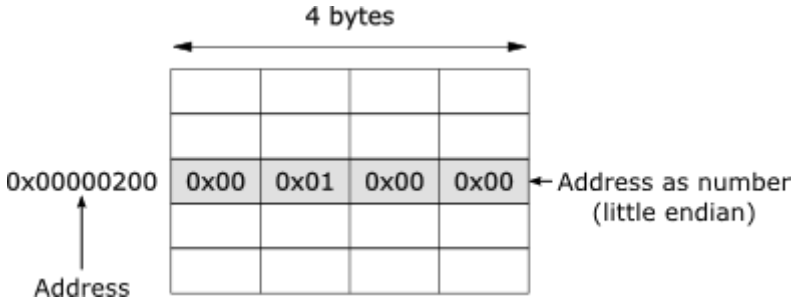


step:

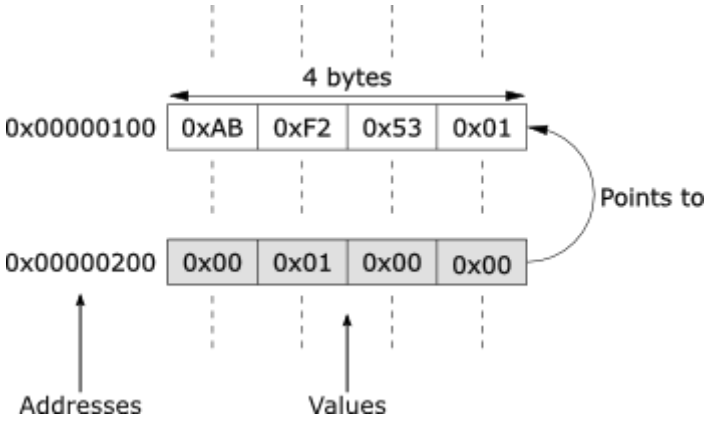
1. Obtain the integer s stored in position d (size)
2. Check that $0 \leq i$
3. Check that $0 < i$
4. Calculate the address of the element as $d + 4 + (t * i)$

Storing memory address

address can be stored in memory as a value, for 32 bits address, need 4 bytes



when want to access the value, take out value from 0x00000200, and use this value as address to read the specific value, this kind of access is called **an indirection**

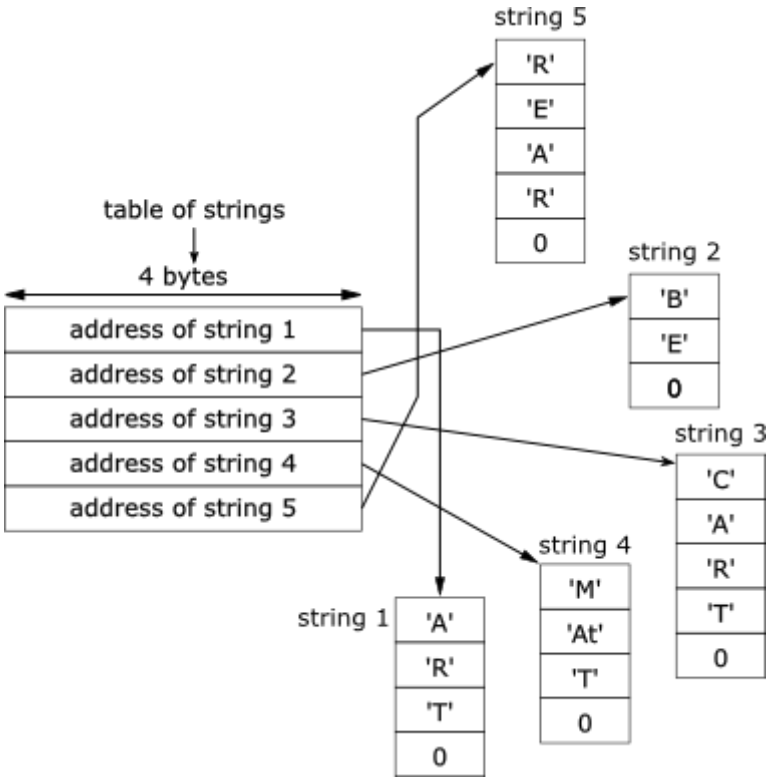


though this chain structure we can establish double indirection, triple indirection and etc

two way to store string

- create a new table with duplicates of all the strings store d in consecutive memory -> waste a lot of space

- create a new table store the addresses of string



Law Name	Expression
Identity	1*x = x, 0 + x = x
Null	0*x = 0, 1 + x = 1
Idempotent	x*x = x, x + x = x
Inverse	x*x' = 0, x + x' = 1
Commutative	x*y* = *y*x, x + y = y + x
Redundancy	x + x'y = x + y, x(x'+y) = xy
Associative	(xy)z = x(yz), (x+y) + z = x + (y+z)
Distributive	x + yz = (x+y)(x+z), x(y+z) = xy + xz
Absorption	x(x+y) = x, x + xy = x
De Morgan	(xy)' = x' + y', (x+y)' = x'y'
Double Complement	(x')' = x

Canonical representations

premise : given truth table

Sum of products:

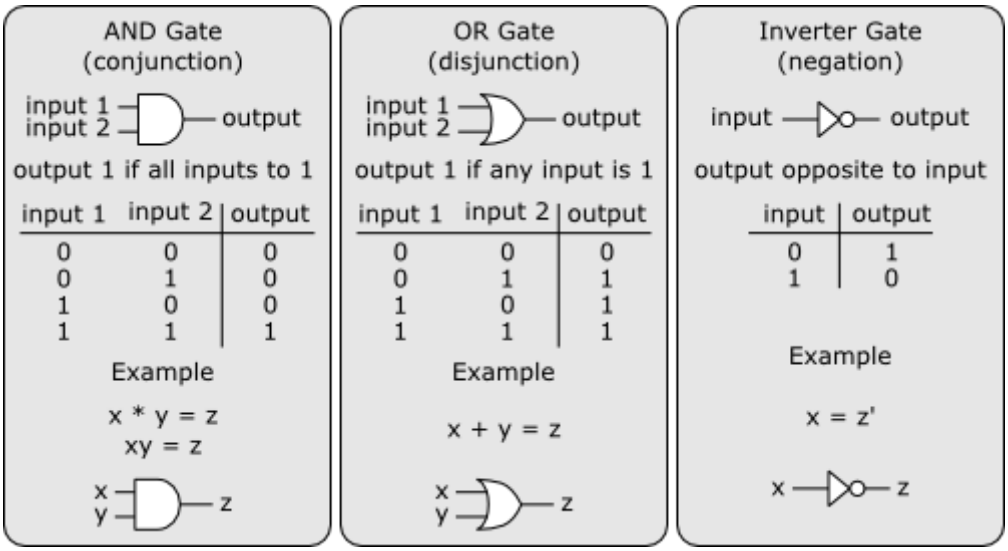
1. select the row which result is equal to 1
2. write the product of symbols and negate those with value 0

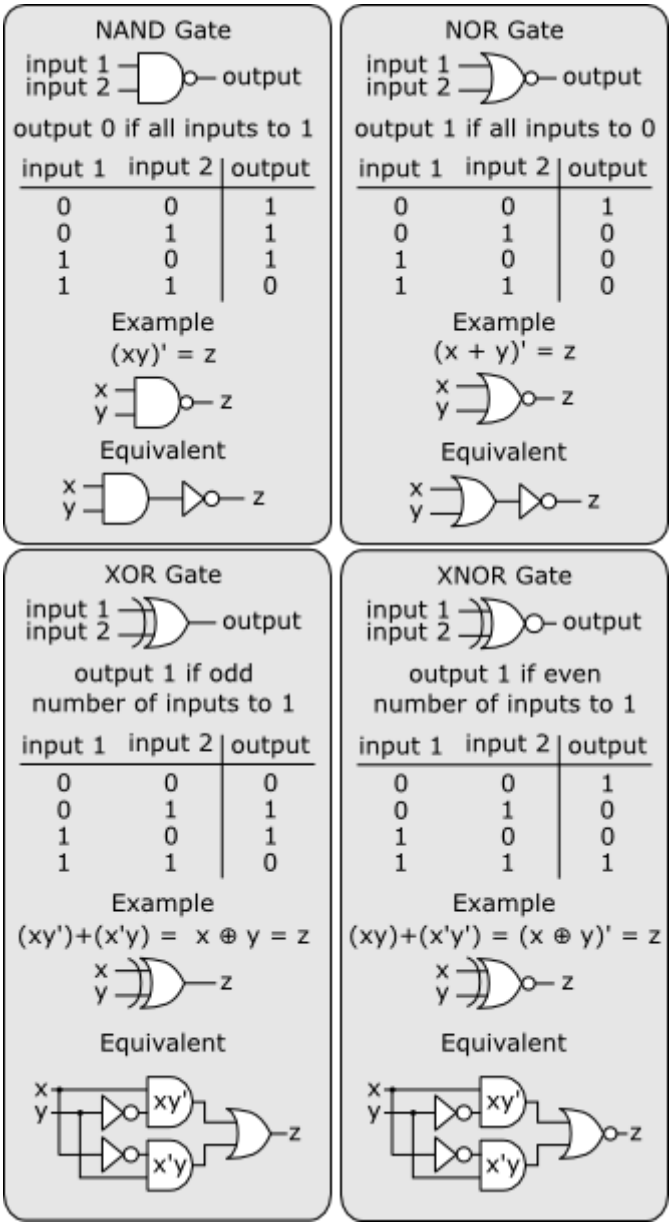
3. add all

Products of Sum:

1. select the row which result is equal to 0
2. write the product of symbols and negate those with value 1
3. multiply all

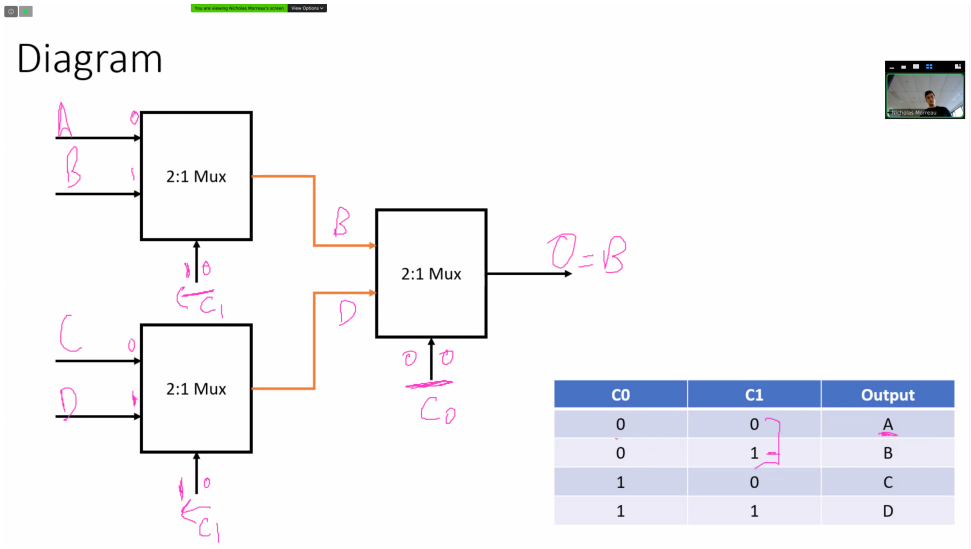
sum of products **equal to** products of sum





mention: XOR and XNOR output depend on the number of inputs, even or odd

Design a multiplexer: use input that can distinguish the output

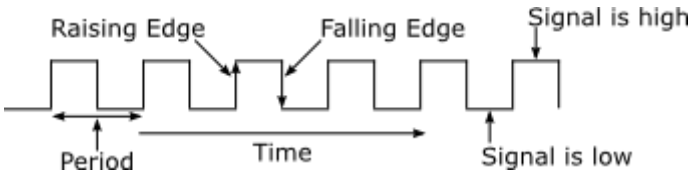


Clock

combinational circuits: the output is either zero or one depending on truth table

sequential digital circuits: the output change depending on the inputs, or remain unchanged over time

the clock -> design when to pay attention to the value of the inputs, when to maintain the value of output



- over time, the value of this signal oscillates from high to low
- the transitions occur almost instantaneously, falling edge, rising edge
- period of clock, represented by T, full transition (from low to high, and back to low), the frequency of a clock represented by f, $f = 1/T$, 5Ghz clock means th signal is making $5 * 10^9$ full transitions per second

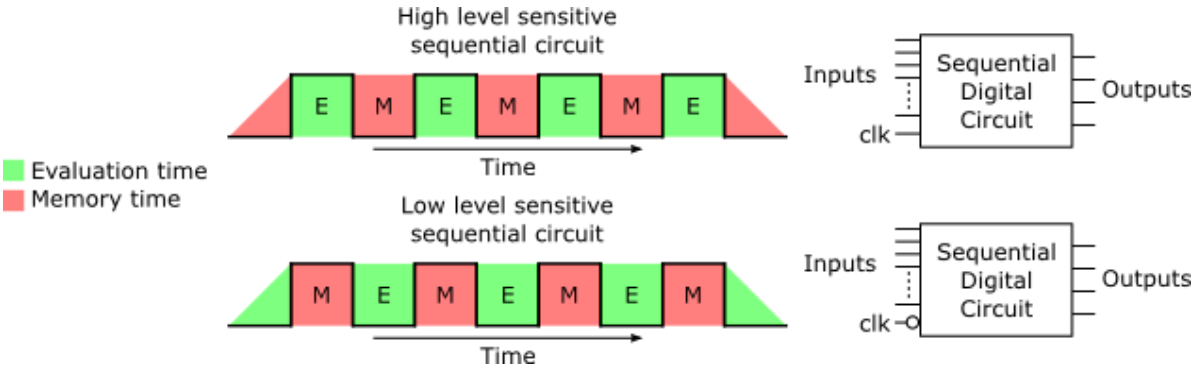
the behavior:

- At a certain instant marked by the clock, behaves as a regular combinational circuit -> **evaluation time**
- At any other times, the value of the inputs are ignored -> **memory time**

Level-Sensitive Sequential Circuits

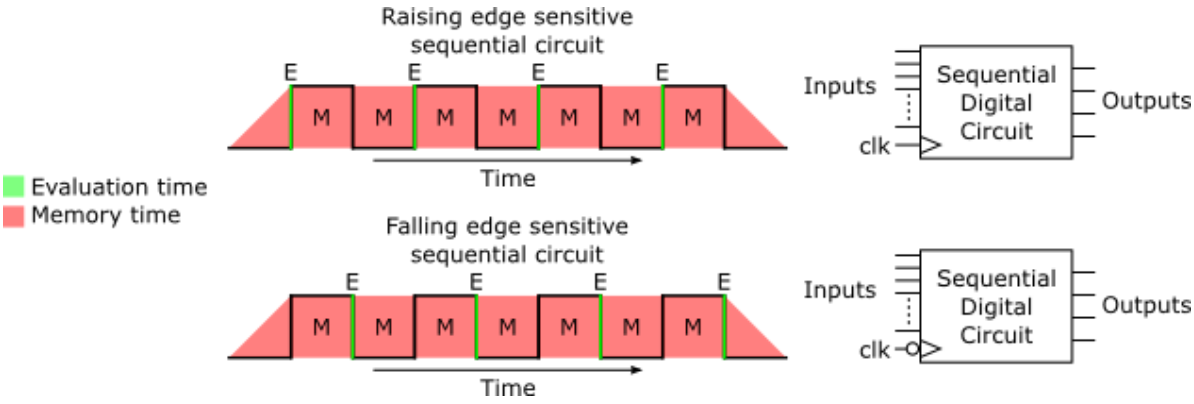
High level sensitive -> when the clock is at a high level, the evaluation time occurs

Low level sensitive

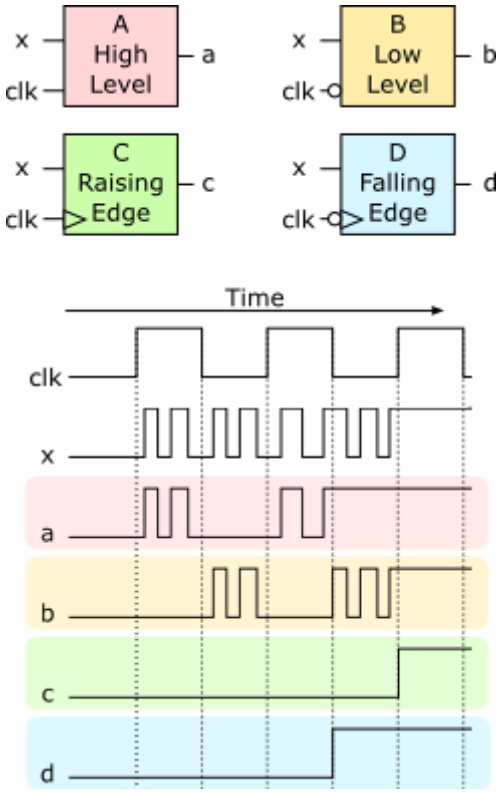


Edge-Sensitive Sequential Circuits

have a shorter evaluation time, evaluate the inputs when the clock makes a transition



Summary:



Filp-Flops

edge sensitive, has one or two inputs (aside from the clock), and two outputs, the two outputs always provide one the opposite value of the other

the truth table's output column is not the immediate value, is the value when the following transition is allowed by the clock.

input	$output_{t+1}$
1	1
0	0

SR Filp-Flop

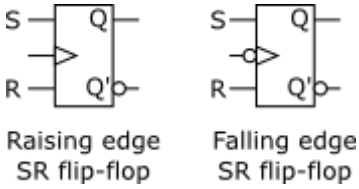
has two inputs，S (set)and R (reset)

Input		Output
S	R	Q_{t-1}
0	0	Q_t
0	1	0
1	0	1
1	1	undefined

when set is high, set output to 1

when reset is high, set output to 0

only makes a transition when allowed by an **edge** in the clock



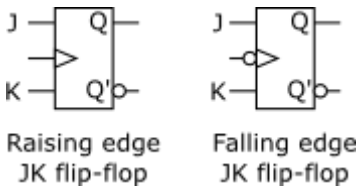
JK Filp-Flop

has two inputs, J (Jump) and K (Knock-Out)

four outputs are defined

Input	Output	
J	K	Q_{t+1}
0	0	Q_t
0	1	0
1	0	1
1	1	Q'_t

when both of inputs are active, flips its output

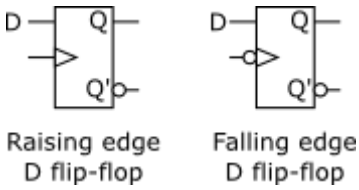


D Filp-Flop

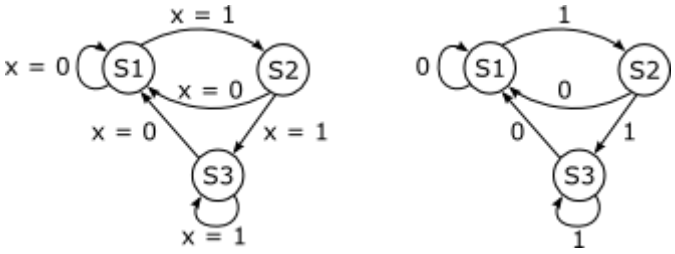
has one input, D

output follow the value of the input when the clock allows a transition

D	Q_{t+1}
0	0
1	1



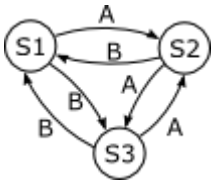
Finite State Machines



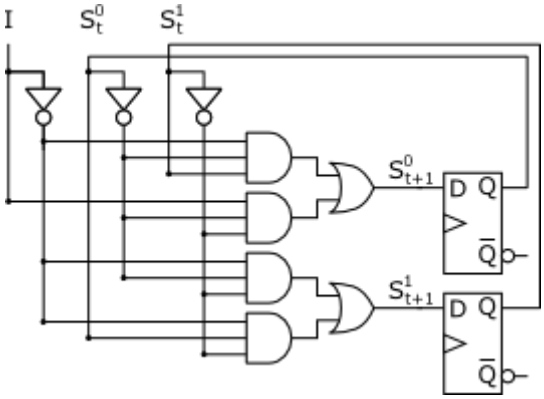
1. identify the different states
2. draw each of the states with a name in a circle
3. identify the system inputs
4. consider how will the state change for every possible combination of input
5. label each transition connecting two states with the input values

convert into a sequential digital circuit

1. identify the possible input values, and the number of states
2. encode the input values using binary logic
3. assign each state a binary combination
4. truth table



Inputs	**Outputs**			
I	S_t^0	S_t^1	S_{t+1}^0	S_{t+1}^1
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	X	X
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	X	X



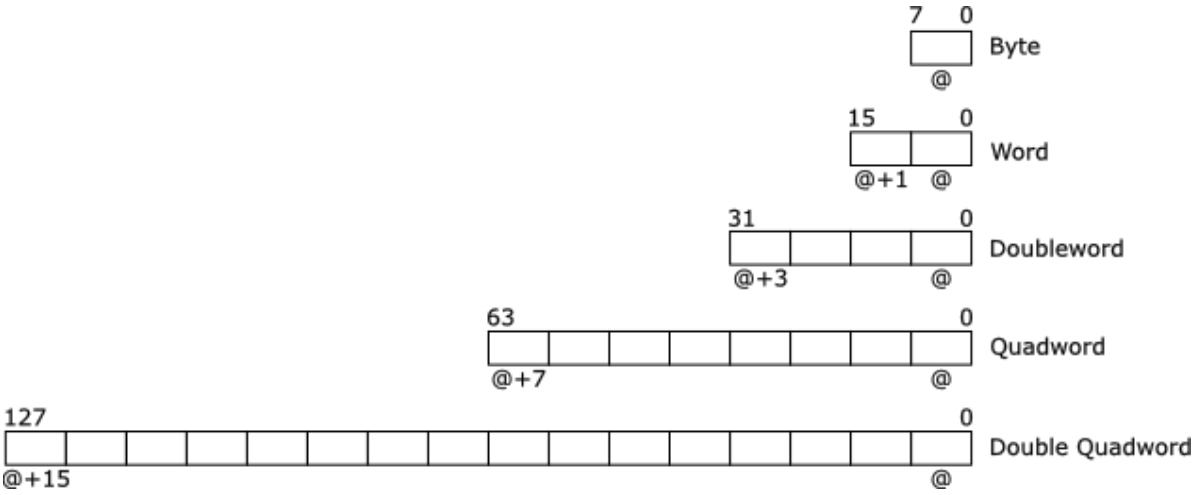
Moore and Mealy

Moore: only depend on the value of the current state, not input

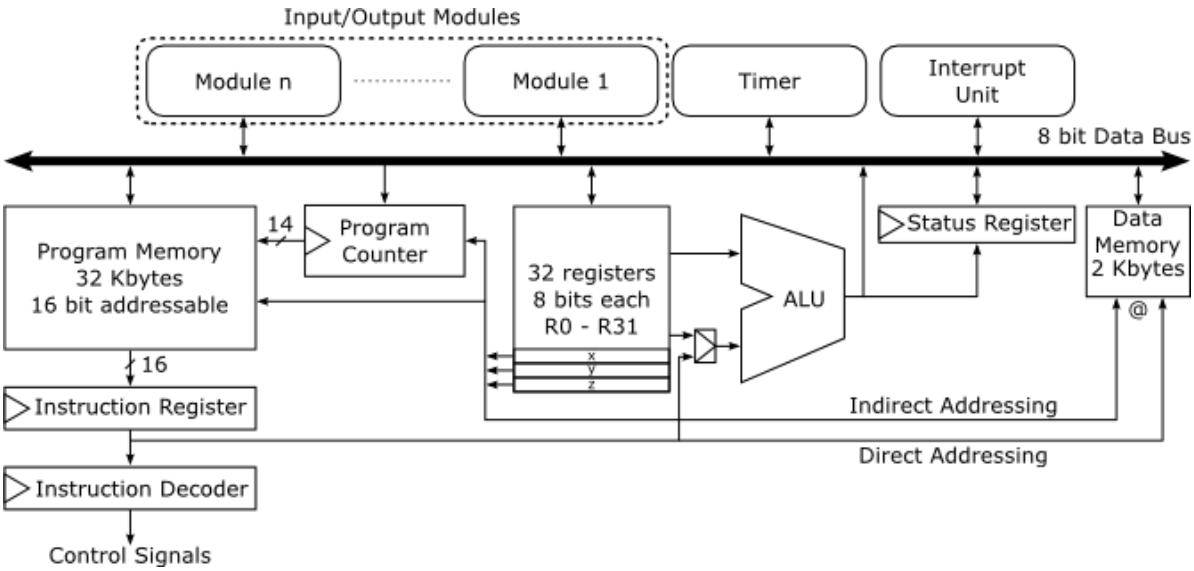
Mealy: depend on both the current state and the value of the inputs

architecture: micro-controllers that are capable of executing the same machine instructions

type of data in AVR micro-controller



the data path



interconnected through an 8-bit data bus

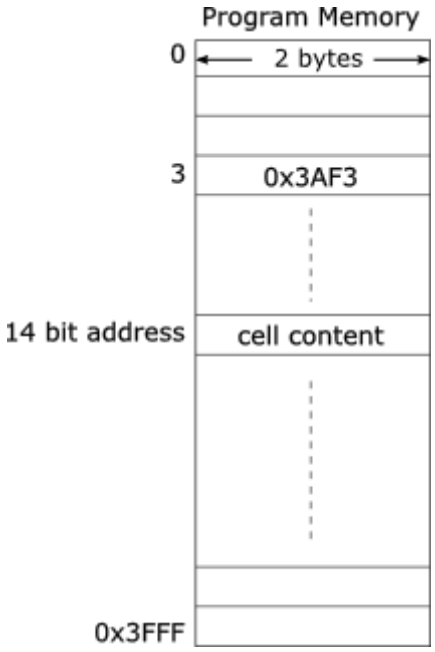
direct Addressing means the operation contains the operand which is indicated to the data memory, indirect addressing means the operation contains the address of registers, have to find the real address in register file.

Memory

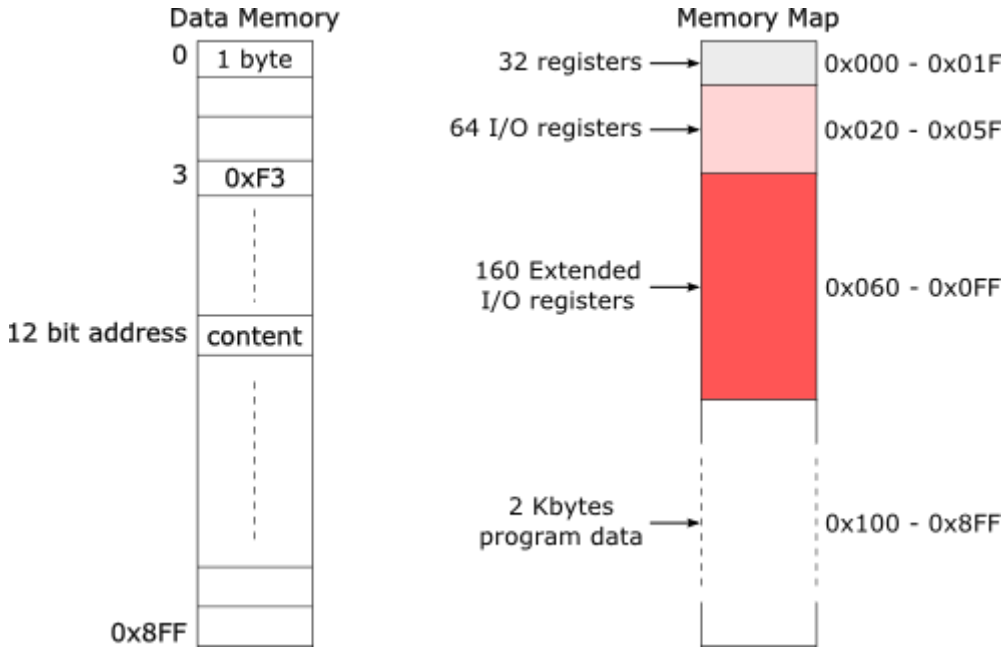
AVR architecture contains two types of memory, **Program Memory**, **Data Memory**

Machine instructions are obtained from the Program Memory and stored in the Instruction Register(decode, sent control signal to the rest of the components)

The program memory is a flash memory, 2 byte addressable (follow picture assume the program memory is 32 kb, 2^{15} byte, since 2 bytes as a cell, 2^{14} is enough to address the memory)



The data memory is a static RAM, 1 byte addressable. (the follow picture assume the data memory is 2 kb, 2^{11} byte, 2048, 0x000 to 0x7FF is enough, but the first 256 position is designed to access 32 general purpose registers, 64 additional I/O register, 160 extended I/O registers)

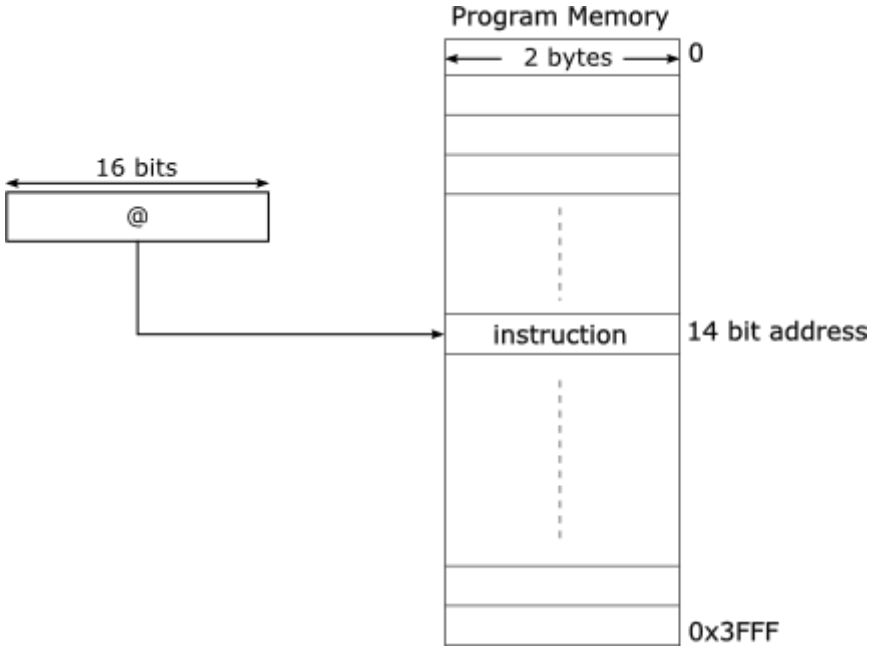


memory mapped input/output: some operation to access register and data

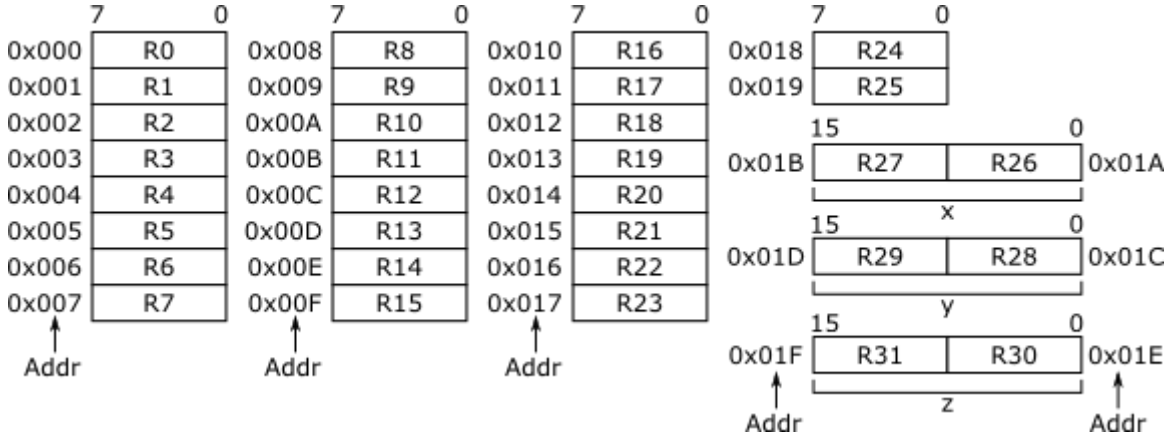
third memory: EEPROM (Electronically Erasable Programmable Read Only Memory)

Register

program counter: contain the address in the program memory of the next instruction to be executed.



General purpose registers: has 32 8-bit, known as **register file**, named with prefix R and a number, from R0 to R31, 1 byte. When 16 bits are needed(to access program memory), use six last registers, can be treated as 16-bit, names X, Y and Z.



Arithmetic/Logic Unit(ALU)

combinational circuit capable of performing operations of three types: arithmetic, logical, bit level functions

operands:

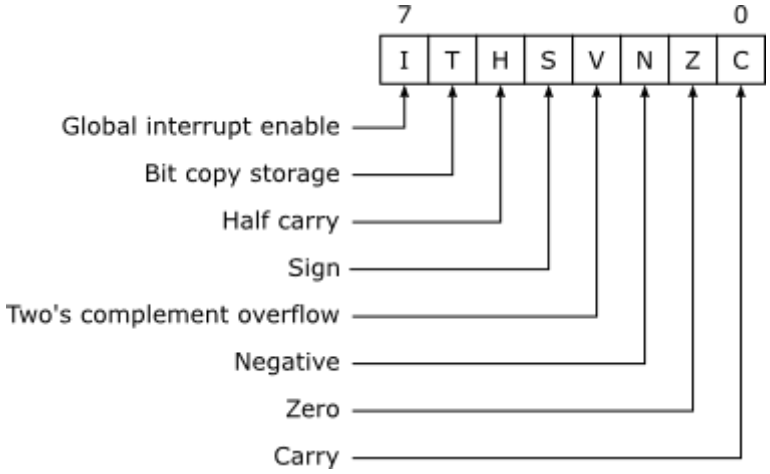
1. from register file
2. one from register file, second from the instrucion register

capable of multiplying two signed or unsigned integers.

Status Register

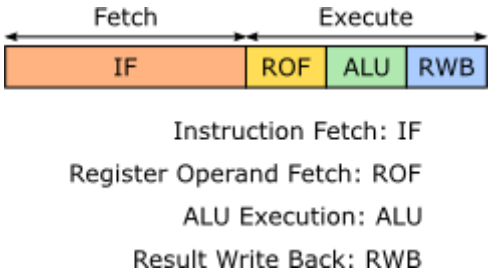
8 bits, reflect special situations when they occur, other instructions make decisions based on these values.

1. Carry flag (C): indicates if a carry has occurred in the latest arithmetic or logic operation.
2. Zero flag (Z): indicates if the result of the latest arithmetic or logic operation has been zero.
3. Negative flag (N): indicates if the result of the latest arithmetic or logic operation has been negative (it is the most significant bit of the latest result).
4. Two's complement overflow flag: (V): if true, indicates that the latest arithmetic or logic operation, if considered over integers encoded in two's complement, has produced an overflow.
5. Sign flag (S): the sign flag indicates the sign of the result of the latest arithmetic or logic operation. It is always the exclusive or between the negative flag and the two's complement overflow flag ($S = N \oplus V$).
6. Half carry flag (H): indicates if a carry has occurred at the 4th bit (half) the operator. Useful for Binary Coded Decimal (BCD)
7. Bit copy storage (T): the source or destination for the bit that is the operand of bit copy operations BST and BLD.
8. Global interrupt enable (I): if set, the interruptions in the microcontroller are processed. If zero, they are ignored.



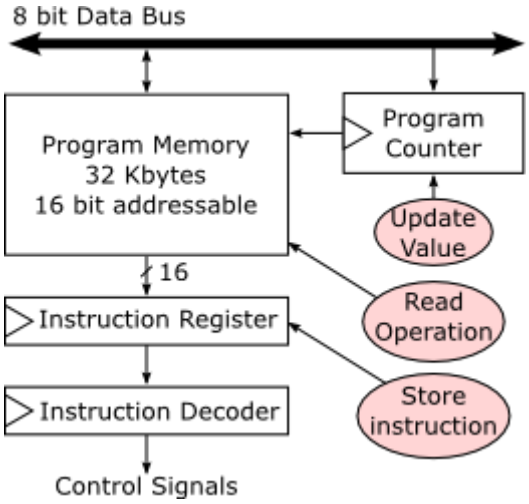
The Execution Cycle

- Instruction Fetch. Obtain the instruction from the program memory and stored in the instruction register.
- Execution. Decoded and executed, divided in following sub-stages:
    - Register Operand Fetch
    - ALU Execution
    - Register Write Back



Instruction Fetch:

- obtains the next instruction from **program memory**, stored in the **program counter** as the address memory, the result of the read operation is stored in the **instruction register**
- the program counter is automatically updated to point to the next memory location



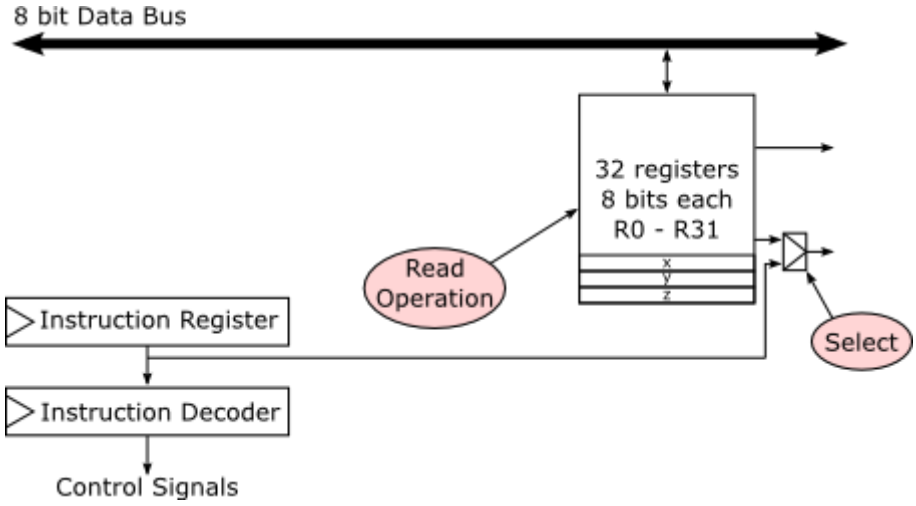
- the controller finds out the type of instruction, controller contains a **sequential digital circuit**
  - receives the instruction
  - generates the appropriate control signals at the next clock cycles
- some operation are 32 bits, require an additional value, in these cases, a second fetch stage is executed

Register Operand Fetch

valid for instructions require the use of the ALU, result written back to the register file.

- The operands are obtained from various sources. Most of the instructions use the values stored in the register file.

  (output from register, input to register)
  - One 8-bit output, one 8-bit result input
  - Two 8-bit outputs, one 8 bit result input
  - Two 8-bit outputs, one 16-bit result input
  - One 16-bit output, one 16-bit result input
- some instructions contain one operand that is not in a register, is part of the instruction. A pre-defined sub set of the instruction bits and use as operand.
- read or write data from the data memory, operands are used to calculate the address of the position where data will be read or written.



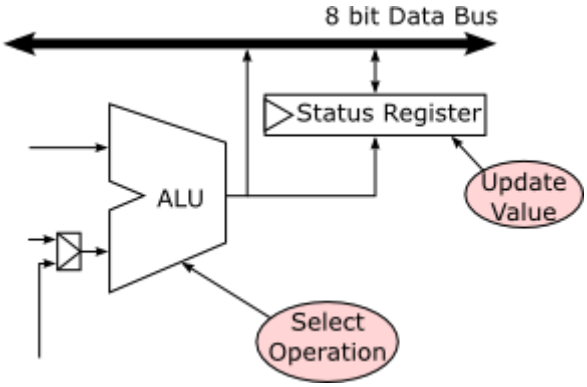
select part is to select the operand is either from registers or from instruction register directly

ALU Operation Execute:

only present in those instructions that require an operation to be performed by the ALU

ALU is a combination circuit.

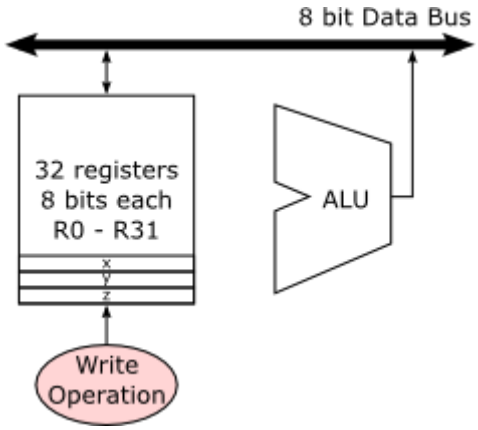
The control signals select the appropriate operation and the result is produce at the output



Result write back

only result produced by the ALU needs to be stored in the register file.

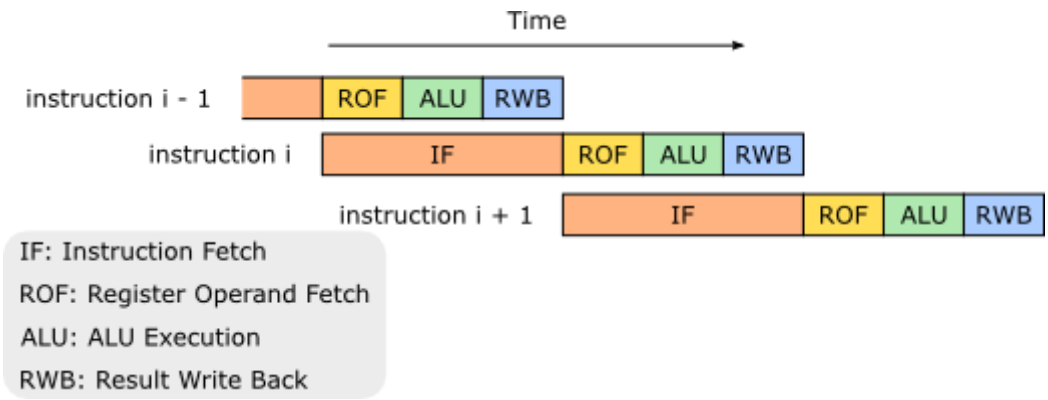
- ALU write result on to the data bus.
- register selects the destination for the result



Pipelined execution

2-stage pipelining

The execution stage of an instruction is done in parallel with the fetch of the next instruction in sequence.



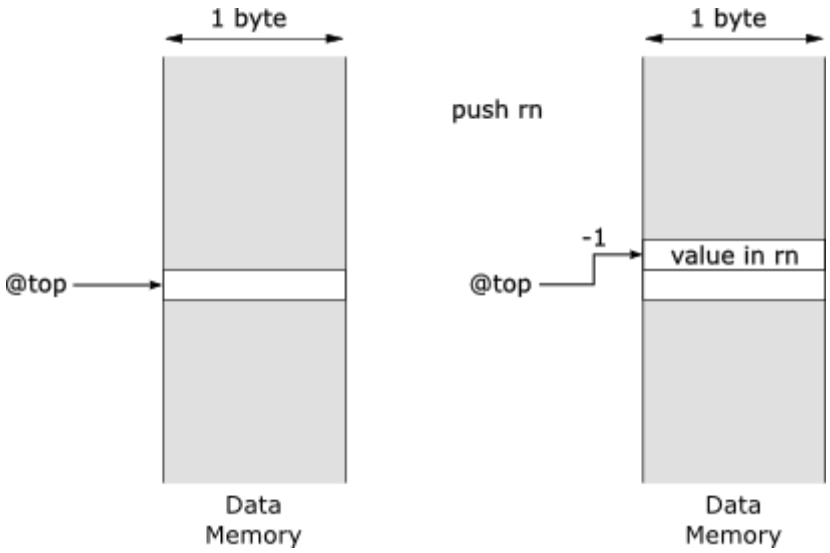
Stack

in AVR, the instructions are restricted to use registers, and the data size is always 1 byte.

push rn

rn is any of the general purpose registers, when the top address is @top:

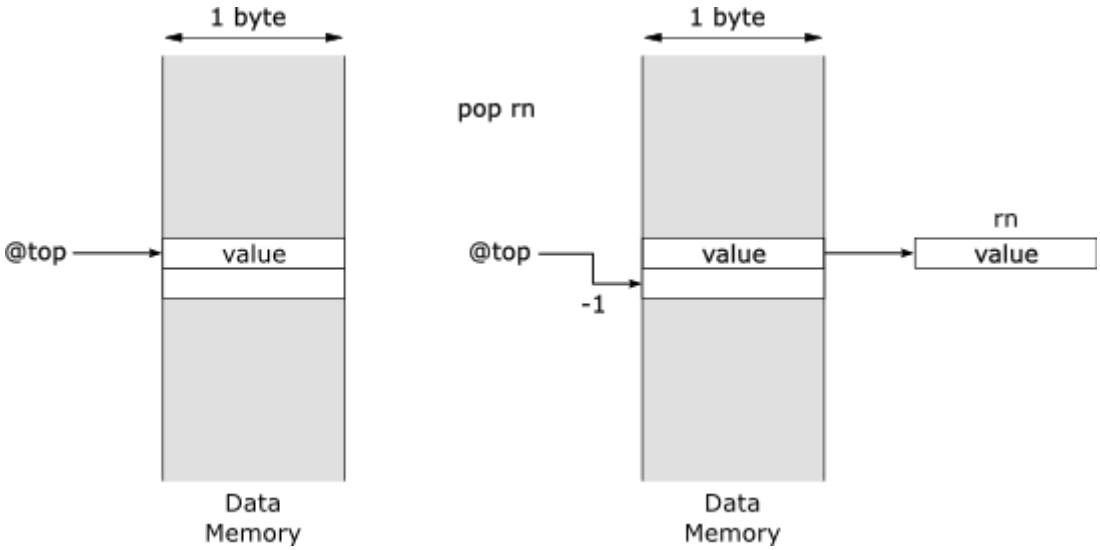
1. @top = @top - 1
2. given register is written in @top



pop rn

store the content at the top of the stack in the register

1. data in @top written in the given registers.
2. @top = @top + 1



- Some processors have machine languages that allow operands of the stack instructions other than the general purpose registers.
- Aside from the stack instructions, the instructions to call and return from a subroutine also use the stack as described in the following table:

Instruction	Description
CALL, ICALL, RCALL	Instructions to call a subroutine. The return address is stored in two consecutive positions in the stack. The stack pointer is thus decremented by two.
RET, RETI	Instructions to return from a subroutine. They **read** the return address from two positions in the stack. The stack pointer is thus incremented by two.

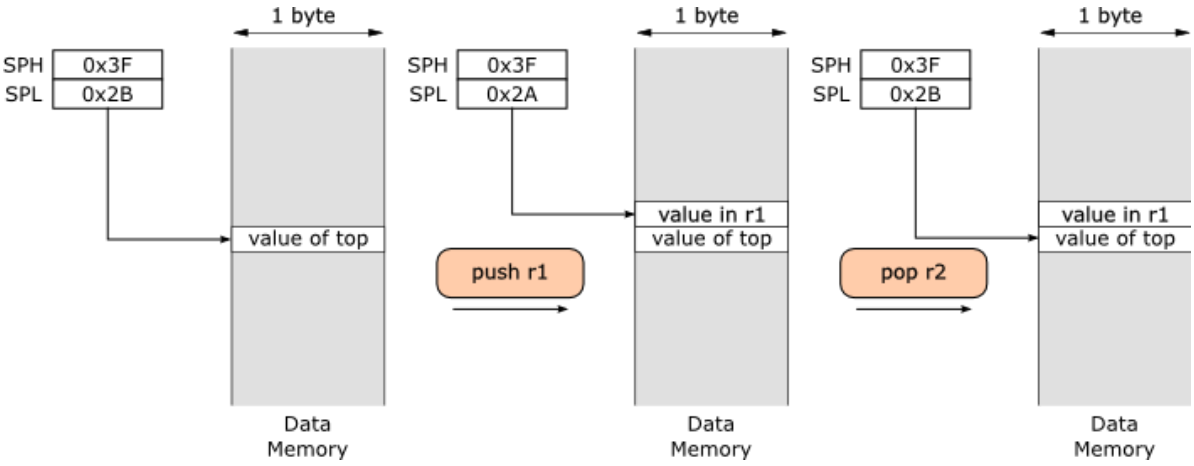
The stack pointer

implicit operand: push and pop doesn't specify the destination.

stack pointer help keeping the address of the top of the stack

in AVR, is a part of generic I/O registers, in 0x3E, 0x3D, **memory addresses** are represented by 16 bits, so use 2 registers

Register name	Address	Content
SPH	*0x3E*	Most significant 8 bits of the address of the top of the stack.
SPL	*0x3D*	Least significant 8 bits of the address of the top of the stack.



Stack Initialization:

1. reserve memory for the stack
2. set the correct value in the stack pointer (typically initialized at th e last position in data memory)

The set of instructions a microprocessor can execute -> **instruction set architecture or ISA**

large number of instructions -> shorter instruction sequences, take longer to execute

small number of instructions -> longer instruction sequences, take shorter to execute

CISC Complex Instruction Set Computers

Rich and complex set of instructions, use several operands and require multiple memory accesses

RISC Reduced instruction set computers

a very simple operation, faster execution, simpler structure

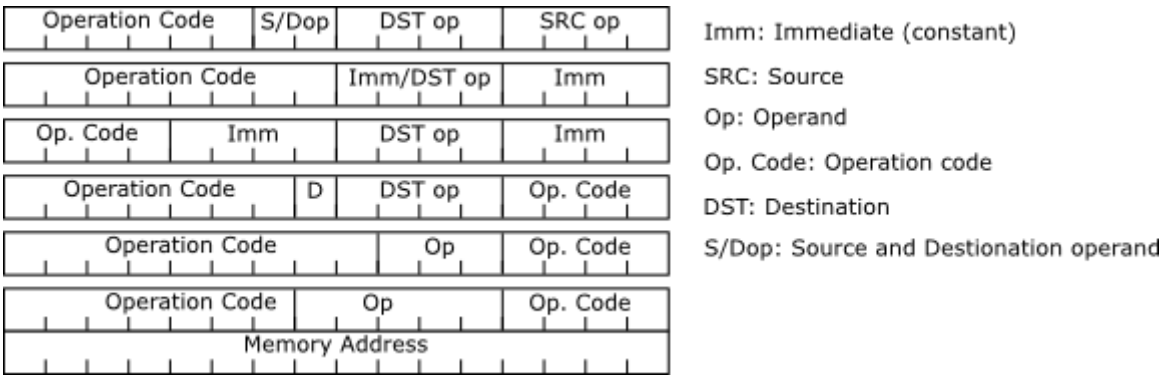
CISC will execute a smaller number of instructions, but each of them will take longer than the RISC architecture

Fixed length format

Variable length format

The AVR architecture has a fixed length format, most of the instructions are encoded with 16 bits

- `Rd`: A register in the register file that will be the destination of the result derived from the instruction.
- `Rr`: A register in the register file which will provide one of the operands for the instruction.
- `R`: Result of the instruction after its execution.
- `K`: A constant value.
- `k`: A constant **memory address**.
- `b`: A bit in a register in the register file or an input/output register.
- `s`: One of the bits of the status register.
- `X`, `Y`, `Z`: 16 bit registers obtained combining two registers in the register file (`X=R27:R26`, `Y=R29:R28`, `Z=R31:R30`).

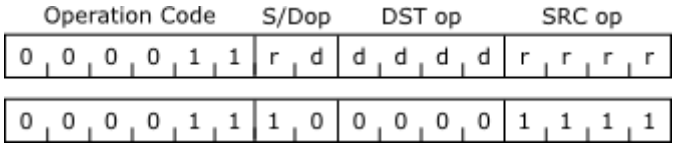


Operation code: every instruction must have some bits to encode the type of operations that is required

---

Instructions for which the operands are one of the 32 general purpose registers, require five bits per operand. These bits are **not necessarily in contiguous positions** in the instructions but this fact makes no difference when decoding the instructions.

```
ADD R0, R31
```



For example, in the top part, five r (1 in S/Dop, 4 in SRC op) indicate to 11111, which means R31, five d indicate to 00000, R0.

---

some instructions allow a number as operand instead of a register

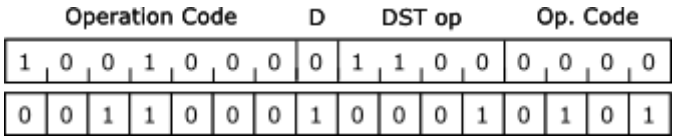
```
CPI R16, 255
```

This instruction is used to compare immediate operands and register, carry out the operation `R16 - 255` and update the status register to reflect the conditions of the result, this instruction can only use 16 of the 32 general purpose registers (index 16 <= d <= 31), because it needs 4 bits to encode the immediate operands.

Exception:

```
LDS R12, 12565
```

For this example, LDS is an instruction store a memory address into register, in AVR, a memory address is 16 bits long, so it need another memory to store the operands



Assembly Language:

same instructions, same operands and formates as the machine language

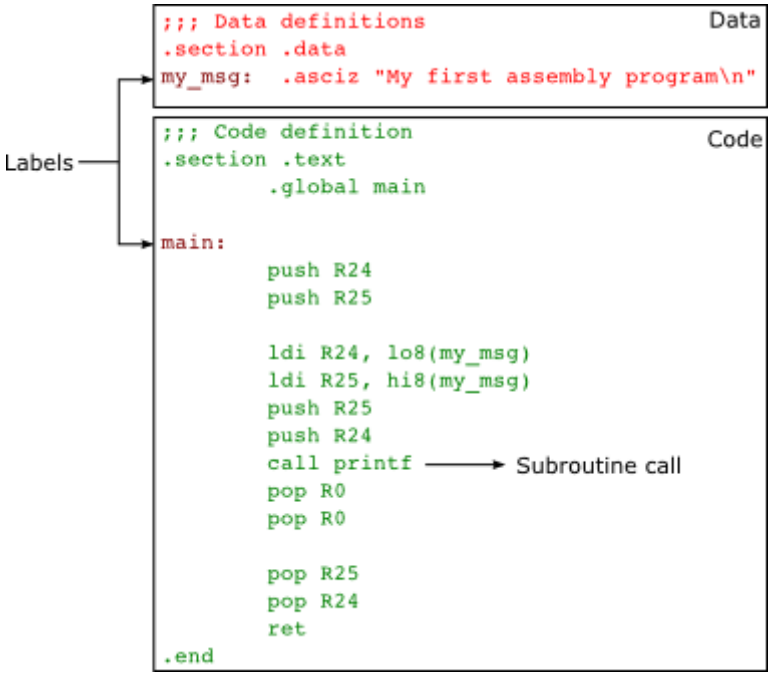
alphanumeric representation, executed bas part of machine language

The translation between assembly language and a machine instruction is a straightforward application of simple encoding rules to transform each operand into a set of bits of the instruction format.

instructions always start with the instruction **mnemonic**, followed by the operands separated by **commas**.

- The first operand after the instruction mnemonic is the **destination** operand.
- The remaining operands are **source** operands.
- Registers are referred by their names which are made by a number between 0 and 31 with the prefix `R` or `r`.
- The assembly instructions are case insensitive. Both `r1` or `R1` refer to the same register, and `add` and `ADD` refer to the same instruction.
- Numeric constants in an instruction are simply represented as numbers, with no prefix. For example `CPI R16, 255`.

- Memory addresses are typically referred by its **label** which must be previously defined in the data section of an assembly program.



`.section .data` doesn't translate into any instruction, just a mark, called **directives**

`my_seg` followed by a colon, this label is defines in an assembly code

`.section .text` tells the assembler that the data definitions have finished

`main` is global symbol

basic structure

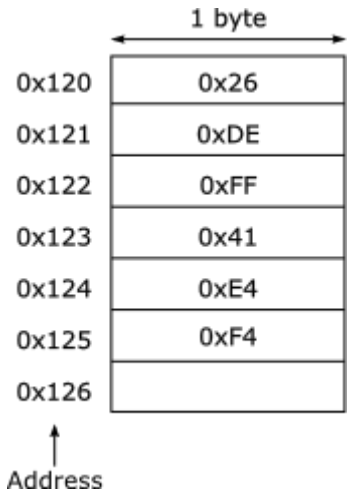
```
;;; Data definitions go here
.section .data

;;; Code definition goes here
.section .text
        .global main

main:
        ret
.end
```

Definition of byte

```
data:    .byte 38, 0b11011110, 0xFF, 'A', 0344, -12
```



`0x120` is chosen arbitrarily

Definition of String

`.ascii` each character is encoded and stored in consecutive memory position

`.asciz` same, but each string is encoded with an extra byte with value `0x00`

`.string` same as above

```
msg:    .ascii "S 1", "S 2"
        .asciz "S 3", "S 4"
        .string "S final"
```

Memory

	1 byte			
msg:	0x53	0x20	0x31	0x53
	0x20	0x32	0x53	0x20
	0x33	0x00	0x53	0x20
	0x34	0x00	0x53	0x20
	0x66	0x69	0x6E	0x61
	0x6C	0x00		

Definition of Space

`.space` reserve memory space with the second value

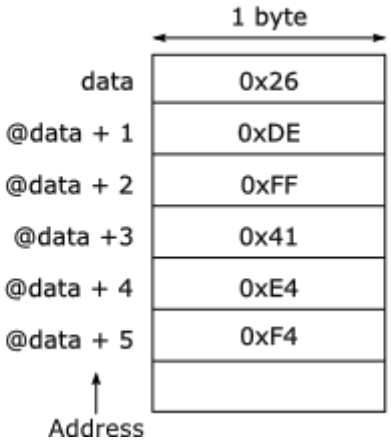
```
result: .space 4, 0
        .space 4
        .space 8, 0xFF
```

Memory

	1 byte			
result:	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0xFF	0xFF	0xFF	0xFF
	0xFF	0xFF	0xFF	0xFF

label

```
data:   .byte 38, 0b11011110, 0xFF, 'A', 0344, -12
```



`hi8()` return 8 most significant bits of the **address** of the label

`lo8()`

- refer to the data
- refer to the address

stack restriction: the top of the stack must be exactly the same before the first instruction and before the last instruction(always `RET`)

Register Restriction:

- R0 is scratch register, need not to be saved nor restored
- R1 always 0
- R2 to R17, R28 and R29, must be saved and restored
- if a function returns an interger, this must be placed in R25:R24 at the end of the subroutine

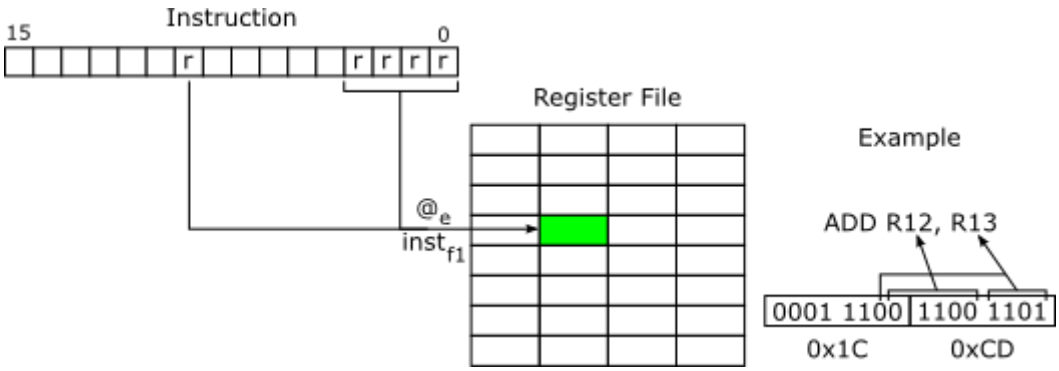
Addressing modes

- Operand are typically stored in general purpose registers, memory or in the instruction
- effective address of an operand will be denoted by `@e`, it can be referred to memory and general purpose register
- the instruction used to calculate `@e` will be denoted by **inst**, stored in `@inst`, this instruction have certain fields to calculate @e, from $inst_{f1}$ to $inst_{fk}$
- If field fi encodes a general purpose register, that register will be denoted by Rfi.
- $DMEM[a]$ refers to the content stored in data memory, in position a, $PMEM[a]$ refers to the content stored in the program memory in position a
- Loading the value v in Register R will be denoted by the expression R ⟵ v.

1. Register Direst

   use to obtain operands stored in one of the general purpose registers in the register file

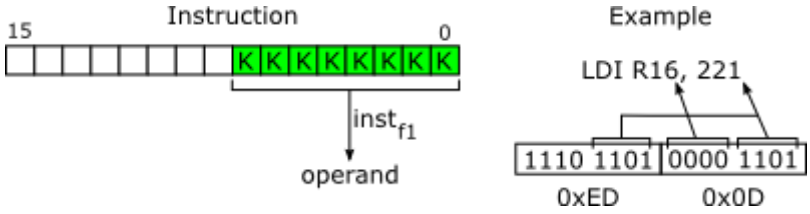
$$@_e = @inst_{f1}$$
$$operand = R_{f1}$$



2. Immediate

   specify a constant as part of the instruction, provides the operand directly, only allow to use registers R16 to R31, four bits available

$$@e = PC$$
$$operand = inst_{f1}$$

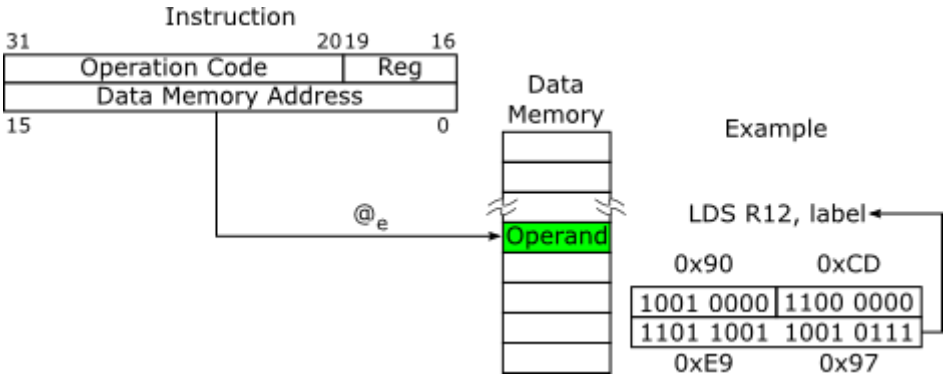


3. Data direct

   access information stored in the data memory with a memory address contained as part of the instruction

$$@e = PMEM[@inst + 1]$$
$$operand = DMEM[PMEM[@inst + 1]]$$

   address is 16 bits, so this kind of instruction is 32 bits

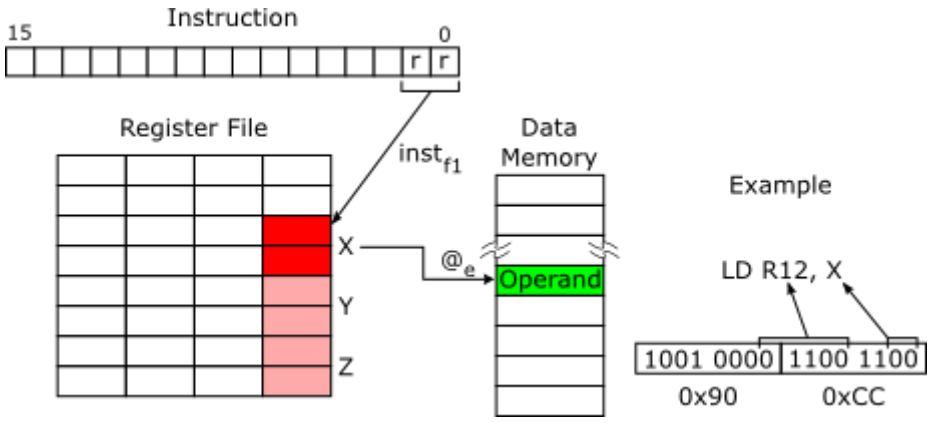


   require three memory access, the program counter need to be updated twice

4. Data indirect

   access an operand stored in the data memory, this address is stored in one of the three 16 bit registers, X, Y, Z. first two bits(two least significant bits) to cover the three options

$$@e = R_{f1}$$
$$operand = DMEM[R_{f1}]$$

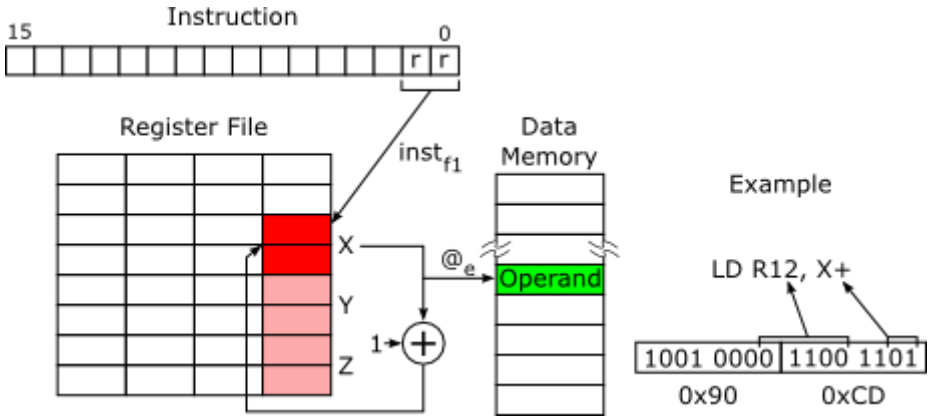


require one additional memory access

---

data indirect with Post-increment

$$@e = R_{f1}$$
$$operand = DMEM[R_{f1}]$$
$$R_{f1} \leftarrow R_{f1} + 1$$



only x, y, z are allowed, the address store in the register is increased, the `+` operation executed as part of the instruction to load the data from memory increases the overall performance of the sequence

`pop` is using this mode

---

data indirect with pre-decrement

$$R_{f1} \leftarrow R_{f1} - 1$$
$$@e = R_{f1}$$
$$operand = DMEM[R_{f1}]$$

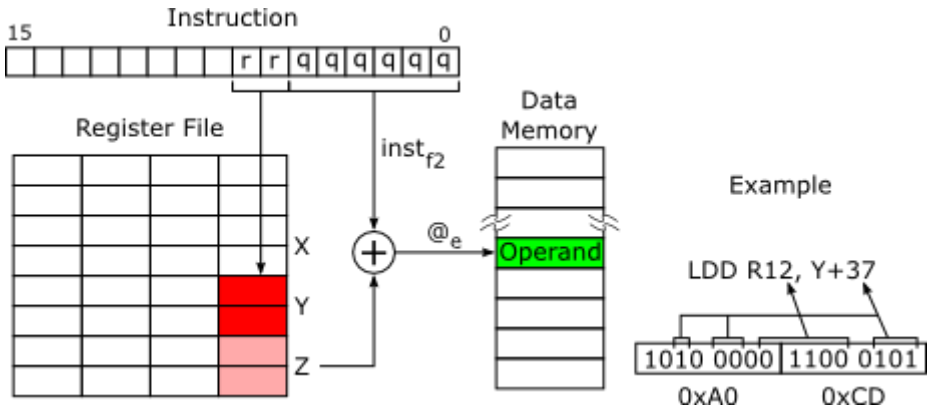
`push`

---

data indirect with displacement

example -> `LDD R12, Y+37`

$$@e = R_{f1} + inst_{f2}$$
$$operand = DMEM[R_{f1} + inst_{f2}]$$



only y and z can be used, the displacement must satisfy 0 <= d <= 63

two way to pass parameters in assembly language:

- through register
- through memory:

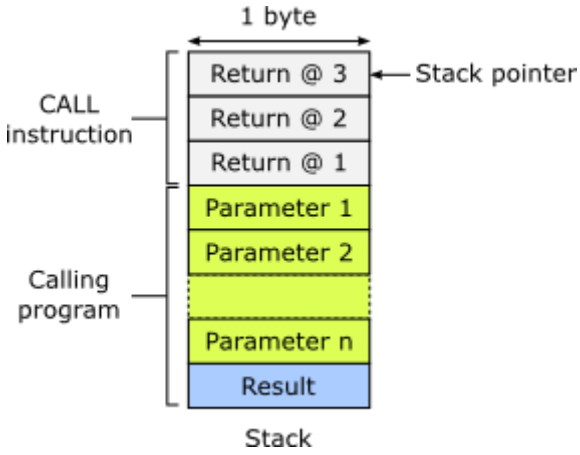
```
p1:     .space 1, 0
p2:     .space 1, 0
result: .space 2, 0

main:
        STS p1, R25     ; Set first param
        STS p2, R24     ; Set second param
        CALL subroutine
        LDI R28, lo8(result)
        LDI R29, hi8(result)
        LD R24, Y       ; Get result
        LDD R25, Y+1

subroutine:
        ...
        LDS R19, p1  ; Get first param
        LDS R18, p2  ; Get second param
        ...
        ...
        ...
        LDI R28, lo8(result)
        LDI R29, hi8(result)
        ST  Y, R24   ; Set result
        STD Y+1, R25
        RET
```

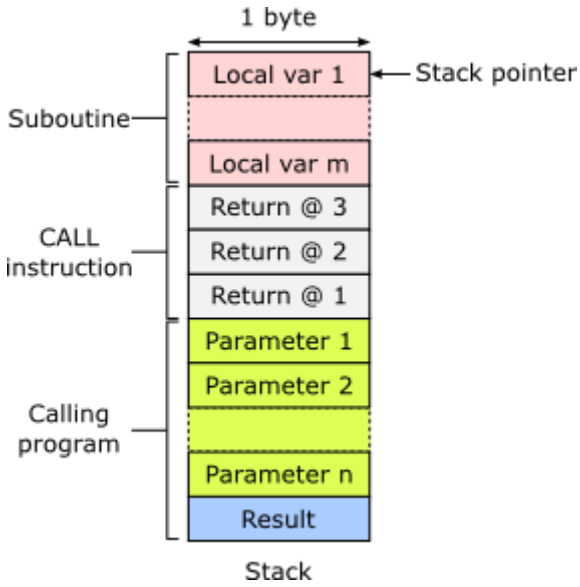
- through stack:

  activation block:



call function automatically push the return address to the stack

```
int translate(Representative r, int radix) {
    int i;      // Local variables
    String str;
    Point p;
    ...
}
```



Calling Program	**Invoked Subrotine**
Reserve stack space to store the result with `PUSH` instructions	Reserve space in the stack for local variables.
Load the parameters in certain order with `PUSH` instructions	Copy the value of the stack pointer in register `Z`
Execute the `CALL` instruction	Execute subroutine code
Remove parameters from the stack with `POP` instructions	Store the result in the location in the stack
Obtain the result form the stack with `POP` instructions	Restore the top of the stack to its initial value.
	Execute `RET`