

# **ELEC 1601**

## **Introduction to Computer System**

### **2016, Semester 2**

Reference: ELEC1601 Course Materials for 2016 Semester 2

#### **Author's Words**

---

This unit requires lots of reading. You should read through the chapter before lecture, attempt all the multiple-choice questions in the reading material, and note down the contents that you do not understand. You should do the tutorial and lecture preparations to test your understanding about the contents, and write down the questions that you did wrong for your future revision.

The following notes summarized the main content that you need to understand to achieve a good grade. Hope my notes can help you, and Good Luck for your studies!

#### **Topics**

---

##### **1. Data Representation**

- 1.1 Different Bases
- 1.2 Binary Encoding
- 1.3 Encoding Integers
- 1.4 Encoding Real Numbers
- 1.5 Encoding Symbols

##### **2. Memory**

- 2.1 Memory
- 2.2 Data Storage
- 2.3 Storing an Array
- 2.4 Storing Memory Address

##### **3. Boolean Logic**

- 3.1 Boolean Expressions
- 3.2 Combinational Circuits
- 3.3 Translations

##### **4. Sequential Digital Circuits**

- 4.1 The Clock
- 4.2 Flip-Flops
- 4.3 Finite State Machines

##### **5. AVR Architecture**

- 5.1 The Execution Environment
- 5.2 The Execution Cycle
- 5.3 The Stack

##### **6. AVR Instruction Set**

- 6.1 Types of Instruction Sets
- 6.2 Instruction Format
- 6.3 The Assembly Language

##### **7. AVR Assembly Programs**

- 7.1 Assembly Program
- 7.2 Guidelines for Assembly Programming

##### **8. Addressing Modes**

- 8.1 Notations
- 8.2 Addressing Modes

##### **9. High Level Programming Constructions**

- 9.1 High Level Programming
- 9.2 Subroutine Execution

# DATA REPRESENTATION

## 1.1 Different Bases

### Numbers Represented in Base b

Properties

- b digits (0 to b-1) are used to represent numbers.
- The number which is 1 larger than b-1 is 10.
- The largest natural number represented with n digits is  $b^n - 1$ .
- Integer multiplication by the base is done by adding a zero as least significant bit. Integer division by the base is by removing the least significant bit, and the least significant bit is the remainder of the division.

### Translations between Different Bases

Between base 10 and other bases

	Method	Example
base b to base 10	$\sum_{i=0}^{n-1} (d_i * b^i)$ where n is the number of digits, and $d_i$ is the digit at position i.	$1234_8 = \sum_{i=0}^3 (d_i * 8^i) = (4*8^0) + (3*8^1) + (2*8^2) + (1*8^3) = 4 + 24 + 128 + 512 = 668_{10}$ .
base 10 to base b	collect the remainders of successive divisions by the base, the digits are obtained from the least to the most significant.	$1234_{10} = 8*154 + 2$ ; $\rightarrow 154 = 8*19 + 2$ ; $\rightarrow 19 = 8*2 + 3$ ; $\rightarrow 2 = 8*0 + 2$ ; The remainders are 2, 2, 3, 2. $\rightarrow$ Result is $2322_8$ .

Between base 8 (prefix 0o) and base 2 (prefix 0b)

Octal digit	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111

Between base 16 (prefix 0x) and base 2

Hexadecimal digit	0	1	2	3	4	5	6	7
Binary	0000	0001	0010	0011	0100	0101	0110	0111
Hexadecimal digit	8	9	A	B	C	D	E	F
Binary	1000	1001	1010	1011	1100	1101	1110	1111

## 1.2 Binary Encoding

Digital circuits manipulate signals in two states: zero and one. Therefore, information processed by digital circuits needs to be encoded using sets of the signals called bits.

A microprocessor is a complex digital circuit. So, all the elements of instructions need to be encoded with binary logic in order for a microprocessor to execute the instructions (machine language).

### Properties of Binary Encoding

Basic Properties

- Number of combinations encoded using n bits: up to  $2^n$  elements.
- Minimal number of bits required to encode N elements:  $N \leq 2^n$  or  $n \geq \lceil \log_2 N \rceil$ .

Rules for Encoding a Set of Elements

- Decide the number of bits used firstly.
- Define a relationship between each element in the set and a concrete sequence of bits.
- Each element must have at least one binary representation.
- Each sequence of bits can only represent one element in the set.

### Size of Encoding

Digital circuits can only manipulate binary numbers with a finite number of bits.

Overflow is the situation in which the result of an operation cannot be represented due to the size of the encoding. The way the microprocessors deal with overflow is simply to detect it and raise an exception so that it can be treated by some specific code.

## 1.3 Encoding Integers

---

### Sign and Magnitude

---

#### Steps

- 1) Encode the absolute value of the number in binary.
- 2) Add the sign as the most significant bit. 0 represents the positive sign, 1 represents the negative sign.

#### Properties

- Range of numbers represented by  $n$  bits:  $[-(2^{n-1} - 1), 2^{n-1} - 1]$ , (i.e. 111...111 to 011...111).
- No. of combination by  $n$  bits:  $2^n - 1$ .
- Zero can be represented by two combinations: 100...000 or 000...000.

### 1s Complement

---

#### Steps

- 1) If the number is positive, simply encode it to base 2.
- 2) If the number is negative:  
Obtain the encoding of the absolute value of the number in base 2.  
Replace every 0 by a 1 and every 1 by a 0.

#### Properties

- Range of numbers represented by  $n$  bits:  $[-(2^{n-1} - 1), 2^{n-1} - 1]$ , (i.e. 100...000 to 011...111).
- No. of combination by  $n$  bits:  $2^n - 1$ .
- Zero can be represented by two combinations: 111...111 or 000...000.

### 2s Complement

---

#### Steps

- 1) If the number is positive, simply encode it to base 2.
- 2) If the number is negative:  
Obtain the encoding of the absolute value of the number in base 2.  
Replace every 0 by a 1 and every 1 by a 0.  
Add 1 to the resulting number.

#### Properties

- Range of numbers represented by  $n$  bits:  $[-(2^{n-1}), 2^{n-1} - 1]$ , (i.e. 100...000 to 011...111).
- No. of combination by  $n$  bits:  $2^n$ .
- Zero has a unique representation: 000...000.
- The most significant bit still represents the sign of the number.
- The addition and subtraction operations follow base 2 rules as if the numbers were naturals.

### 2s Complement to base 10

---

Translate an integer in 2s complement to representation in base 10.

#### Steps

- 1) If the number is positive (i.e. the most significant bit is 0), translate it to number in base 10 directly.
- 2) If the number is negative:  
Replace every 0 by a 1 and every 1 by a 0.  
Add 1 to the resulting number.  
Translate the resulting number to base 10 and take its value as a negative number.

## 1.4 Encoding Real Numbers

---

There are infinite real numbers in an interval, so the representation of real numbers must be restricted to a certain interval and to certain values within that interval.

Manipulating real numbers by a digital circuit may introduce an error, because the result of arithmetic operations on real numbers can only be approximated to the closet number that can be represented.

## Floating-Point Encoding

---

### Steps

- 1) Represent the real number in base 2, with one set of digits to the left of the decimal point, and the other set of digits to the right set of the decimal point. Note that the weight of the digits in the decimal part is negative powers of the base, i.e.  $2^{-1}$ ,  $2^{-2}$ ,  $2^{-3}$ , etc.
- 2) Adjust the value of the mantissa so that the first non-zero digit is at the right of the point, and it is multiplied by the appropriate power of the base.
- 3) Encode the three parts separately: the sign of the mantissa, the mantissa, the exponent. The mantissa is already in binary, encode the exponent using any methods (i.e. Integers encoding).

Example: 5.875

- 1)  $5.875 = 2^2 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3}$ , in binary is: 101.111
- 2)  $101.111 = 0.101111 * 2^3$
- 3) Sign: 0; Mantissa: 101111; Exponent:  $3_{10} = 011_2$ . → Encoding: 0 101111 011.

Note: Representation of zero using float-point encoding is a special case.

Note: The most significant bit of the mantissa is always 1.

## Range

---

Suppose we use 1 bit for the sign,  $a$  bits for the mantissa,  $b$  bits for the exponent.

- Range:  $[-0.11\dots1 * 2^{(2^{(b-1)}-1)}, 0.11\dots1 * 2^{(2^{(b-1)}-1)}]$
- Negative numbers  $\in [-0.11\dots1 * 2^{(2^{(b-1)}-1)}, -0.1 * 2^{(-2^{(b-1)})}]$
- Positive numbers  $\in [0.1 * 2^{(-2^{(b-1)})}, 0.11\dots1 * 2^{(2^{(b-1)}-1)}]$

Example: use 1 bit for sign, 7 bits for the mantissa, 6 bits for the exponent.

- Negative numbers  $\in [-0.1111111 * 2^{31}, -0.1 * 2^{-32}]$
- Positive numbers  $\in [0.1 * 2^{-32}, 0.1111111 * 2^{31}]$

## Accuracy and Precision

---

Accuracy refers to how close a binary encoding of a real number is to the real magnitude we want to encode.

Precision is a property of the overall encoding scheme.

- For numbers represented by the same number of bits, the more bits used for mantissa, the higher precision of the encoding.

## Overflow and Underflow

---

Overflow and underflow are possible exceptions when operating with real numbers.

- *Overflow*: When the number to be represented is outside of the range of valid values or when trying to represent a very small number.
- *Underflow*: When approximating a non-zero number by zero, such as when dividing a very small number by natural numbers.

## 1.5 Encoding Symbols

---

The three ingredients:

- A set of symbols.
- Size of encoding (the number of bits required).
- The correspondence between each symbol and the sequence of bits representing it.

## Encoding Characters

---

Encoding Schemes:

- ASCII (American Standard Code for Information Interchange): 8 bits, 256 combinations.
- EBCDIC (Extended Binary Coded Decimal Interchange Code).

UNICODE: UTF-8, UTF-16, UTF-32.

## Encoding Instructions

Note: For all following types of encoding, there are a minimum number of bits required for encoding. When more bits are used, the extra filler bits with value zero are put on the right side of the code.

### UAL-1:

The instructions are divided into three parts: an operation and two operands that are integers in the range [0, 255].

- Structure: 

Operation (2 bits)	8 bit number	8 bit number
--------------------	--------------	--------------
- The operation values: add (0b00), sub (0b01), mul (0b10), div (0b11).
- At least 18 bits are needed to encode the instructions, the number of elements is  $2^{18}$ .
- Example: Instruction "add 0x27 0xB2". → Encoding: 00 0010 0111 1011 0010 000000 → 0x09EC80

### UAL-2:

The instructions are divided into four parts: an operation, two operands that are integers in the range [0, 255] and an operand to represent the location to store the result of the operation.

- Structure: 

Operation	8 bit number	8 bit number	Location (1 bit)
-----------	--------------	--------------	------------------
- The operation values: add (0b00), sub (0b01), mul (0b10), div (0b11).
- The location has two possible values: Location\_A (0b0), Location\_B (0b1).
- At least 19 bits are needed to encode the instructions, the number of elements is  $2^{19}$ .

### UAL-3:

Similar to UAL-2, but the two operands can now be one of the two locations (i.e. Location\_A, Location\_B), where both locations can be the place where operands are stored.

- Structure: 

Operation	9 bit operand	9 bit operand	Location (1 bit)
-----------	---------------	---------------	------------------
- Encode the operand which is an integer: 0 as the first bit + 8 bits encoding the number.
- Encode the operand that is a location: 1 as the first bit + 7 bits ignored + 1 bit representing the location.
- The number of elements is  $4 * (2^8 + 2) * (2^8 + 2) * 2 = 532512$ .
- Example: "mul Location\_B 0x04 Location\_A" → Encoding: 10 100000001 000000100 0 000 → 0xA02040.

# MEMORY

## 2.1 Memory

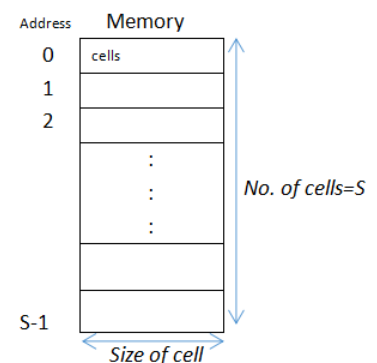
### Random Access Memory (RAM)

Definition: the storage location in the computer system to store and remember information.

- RAM is volatile, as when the power is turned off, the data is lost.
- The memory cells/addresses are numbered started with zero.
- The addresses are encoded in base 2. The relationship between the size S of a memory chip and the number of bits  $n$  used to encode the addresses:  $S \leq 2^n$ .

Units:

Prefix	kilo	mega	giga	tera	peta	exa	zetta	yotta
Symbol	K	M	G	T	P	E	Z	Y
Size	$2^{10}$	$2^{20}$	$2^{30}$	$2^{40}$	$2^{50}$	$2^{60}$	$2^{70}$	$2^{80}$



### SRAM and DRAM

Static RAM (SRAM)

- Data is stored in a cell until it is overwritten or the power is stopped.

### Dynamic RAM (DRAM)

- The structure of the cell is much simpler, and the value is stored using capacitors. If the capacitor is charged, it stores a one, else it stores a zero.
- Simpler cells → memory chips with higher capacity (i.e. number of bytes) are easily implemented → Low cost and high capacity.
- The capacitors leak current → charge in the capacitors disappears → the cell will lose value → need a read operation over the cell to refresh the charges.
- Each cell must be read after certain time interval → slowing down the overall read/write operations → need to include a mechanism to refresh the cells by reading all of them continuously before the capacitors lose charge.

Computer systems use both types of memory. For components that need to be extremely fast, SRAM is used. For those where a large amount of memory is needed, DRAM is used.

### Memory Operations

#### Read and Write Operations

- *Read Operation*: the memory receives an address and returns the content of the cell with that address.
- *Write Operation*: the memory receives an address and a value, it then writes the value in the cell with the given address, with no result produced.
- Note: The data initially stored in memory before any operation is executed is undefined. If the first operation is a read operation, the result is undefined.

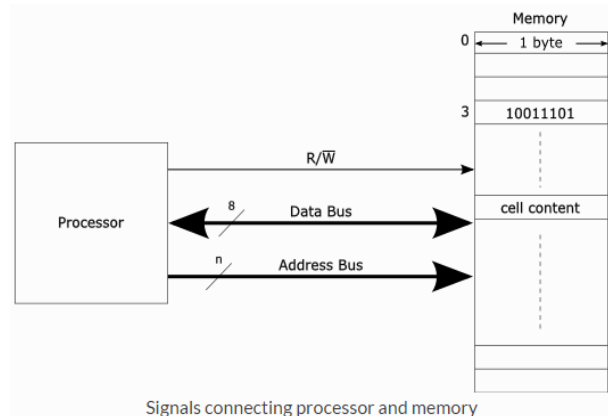
Problem: Time taken for a memory operation is much larger than the speed of the processor, causing the overall system being delayed.

- Solution: To perform read and write operations for several cells at the same time.
- How: Using four memory chips connected in parallel, the cells with address  $4k$ ,  $4k+1$ ,  $4k+2$ ,  $4k+3$  is stored in the first/ second/ third/ forth chip. → Four consecutive cells are stored in different modules → An operation over four consecutive cells will take the same time as it takes one module to perform the operation.
- Requirement: the blocks of four cells have to be aligned with addresses that are multiple of 4.
- Drawback: there might be some operations that manipulate more cells than what is really required.

### Connection between Memory and Processor

**Bus**: a set of wires in which several circuits can read and write binary values.

Bus	Content Sent	Direction
$R/\overline{W}$	Type of operation. 1 encode read, 0 encode write.	from the processor to the memory chip
Address bus	The address of the operation	from the processor to the memory chip
Data bus	The data of a write operation	from the processor to the memory chip
	The data of a read operation	from the memory chip to the processor



For the example in the diagram, the data bus has 8 bits, the address bus has  $n$  bits. Hence, the maximum memory size for this computer system is  $2^n$  bytes.

## 2.2 Data Storage

A simple rule to store data in memory is to use as many consecutive cells as needed to store a complex data structure. The address of the first cell from which the data structure is stored is the data address.

When store data in memory, we need to note the address of the first cell and the size of storage.

## Booleans

---

Values: true, false

Size: 1 bit

Storage

- **Method 1:** Eight Boolean values stored in a 1 byte memory cell.  
Disadvantage: to access the value, need to know the memory address and the position of the bit inside the byte.
- **Method 2:** Boolean is stored in an entire memory cell, leaving the rest of bits untouched.  
Disadvantage: 7 bits are wasted.  
Advantage: access to the value is much faster.

## Characters

---

Value: character

Size: 16 bits / 8 bits

The storage depends on the encoding scheme used

- ASCII: 8 bits for each character → if memory cell size is 1 byte, 1 cell is needed per character.
- Unicode, UTF-16: 16 bits for each character → 2 consecutive memory cells for one character.

## Integers and Natural Numbers

---

Two parts to know:

- Size of the representation.
- Order in which the bytes are arranged. (i.e. Big endian/ Little endian).

Storing Order

- *Big endian*: the first byte stored is the most significant. (E.g. for 0x12345678: 0x12, 0x34, 0x56, 0x78).
- *Little endian*: the first byte stored is the least significant. (E.g. for 0x12345678: 0x78, 0x56, 0x34, 0x12).
- Each computer system uses one of the two methods to store and manipulate integer.
- When two processors in a system use different policies, manipulating integers and naturals requires additional operations to swap the order in which the bytes are manipulated.

## Machine Instructions

---

Two parts to know:

- Type of the processor. (i.e. Fixed length instructions / Variable length instructions).
- Sum of size of the instructions.

Length of Binary Encoding of Instructions

- **Type 1:** Fixed length for all instructions.  
Feature: Accessing the instructions in a sequence is very easy. However, a small number of operands are allowed in every instruction, and the number of different instructions is reduced.
- **Type 2:** Variable instruction length.  
Feature: An instruction may have an arbitrary number of operands. However, during decoding stage, the address of next instruction in a sequence needs to be deducted from the current instruction's address and size.

## 2.3 Storing an Array

---

Let the table contains  $n$  elements, and each element with a size of  $m$  bytes.

- Index of table is from 0 to  $(n-1)$ .
- The table requires at least  $n*m$  bytes.
- If the elements are stored from the address  $a$  → address of element at position  $i$  of the table:  $\text{address}(\text{table}[\text{index}]) = a + (m * \text{index})$ .

To know the number of elements in a table:

- **Technique 1:** Use an extra special value (i.e. zero) after the last element to mark the end of the table. Used when storing a table of characters (or a string).  
Note that it only works if there is a special code for the end of sequence.
- **Technique 2:** To store the size of the array in another memory location as a natural number.

table	address
table[0]	$a$
table[1]	$a + m$
table[i]	$a + (m * i)$
← m bytes →	

## Storing arrays in Java

The Java language stores the size of the array in the **first** memory positions of the table.

Before accessing an element, the program executes additional instructions to verify that the index value is valid, else the `ArrayIndexOutOfBoundsException` exception is raised.

For table  $t$  where elements (each  $m$  bytes) are stored starting at position  $a$ . To access an element in position  $i$ , the program performs the following operations:

- 1) Obtain the size of the table  $s$  stored in position  $a$ .
- 2) Check that  $0 \leq i < s$ . If not satisfied, raise exception.
- 3) Calculate the address of the element  $t[i]$  as  $a + 4 + (m*i)$ .

## 2.4 Storing Memory Address

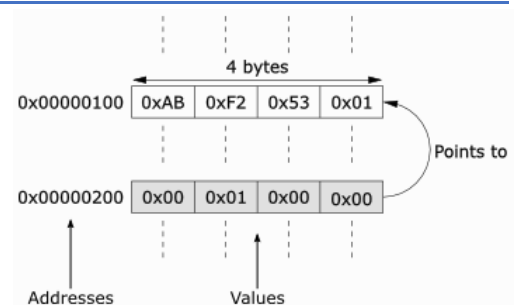
A memory address can be considered as a natural number, and be stored in the memory.

### Indirection

Indirection: The value  $v_1$  stored in a position is the address of another value  $v_2$ , we say that  $v_1$  indirectly points to  $v_2$ .

**Example:** In the diagram on the right.

- Not access the number 0x0153F2AB directly.
- We firstly obtain the address of the number 0x00000100 from another address 0x00000200, and then find the number at the address 0x00000100.
- 0x00000200 is the address of the address of value 0x0153F2AB.



The indirection technique can be chained an arbitrary number of times (i.e. Multiple indirection). The address of the position that contains the address of the data can itself be stored in memory.

# BOOLEAN LOGIC

## 3.1 Boolean Expressions

### Boolean Algebra

Values:  $\{0, 1\}$  or  $\{\text{false}, \text{true}\}$

Operations:

	Generic Name	Symbol	Result
Conjunction	AND	$a * b$	the result is one/true when both operands are one/true
Disjunction	OR	$a + b$	the result is one/true if any operand is one/true
Negation	NOT	$a'$ or $\bar{a}$ or $\neg a$	the result is the opposite of the operand

Boolean expressions can be arbitrarily complex and contain arbitrary operations among symbols, but only using the three operations: conjunction, disjunction and negation. E.g.  $x * y + z'$ .

### Truth Table

A truth table contains all possible combinations of values for the symbols that appear in an expression and the result of evaluating the expression.

- One column for each symbol in the expression, one column for the evaluation of the expression.
- Each row presents one possible combination of the values of its symbols.
- Number of rows =  $2^{\text{no. of inputs}}$



Example: Truth table for  $ab + a$

a	b	Result
0	0	0
0	1	0
1	0	1
1	1	1

For two Boolean expressions over the same set of symbols, they are *equivalent* if for every possible combination of values of its symbols, they produce the same result, or have identical truth table.

### Simplification of Boolean Expressions

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

Example:  $(x + y)(x + y')$

$$(x + y)(x + y') = xx + xy' + xy + yy' = x + xy' + xy + 0 = x + x(y' + y) = x + x = x$$

When a simplification process transforms a three-symbols (x, y, z) Boolean expression into a simpler expression with two symbols (x, z). In this case, y is a *redundant variable*.

### Canonical Representations

Representations: Sum of products and Product of sums.

Function: to find the Boolean expression from a truth table.

Note:

- The SOP expression and the POS expression derived from the same truth table are equivalent.
- No. of products in SOP + No. of sums in POS = No. of combinations.

#### Sum of Products

- 1) Select from the truth table only the rows with the result equal to one.
- 2) For each row selected, write a product with all the symbols. If the symbol has a value of 0, the symbol appears negated; otherwise, the symbol appears without negated.
- 3) Add all these products together in a disjunction.

Example:

x	y	z	OUT	Product
0	0	0	1	$x'y'z'$
0	0	1	0	
0	1	0	1	$x'yz'$
0	1	1	0	
1	0	0	1	$xy'z'$
1	0	1	0	
1	1	0	1	$xyz'$
1	1	1	1	$xyz$

Sum of Product, SOP:  $x'y'z' + x'yz' + xy'z' + xyz' + xyz$

### Product of Sums

- 1) Select from the truth table only the rows with the result equal to zero.
- 2) For each row selected, write a sum with all the symbols. If the symbol has a value of 0, the symbol appears without negated; otherwise, the symbol appears negated.
- 3) Multiply all these sums together in a conjunction.

Example

x	y	z	OUT	Sum
0	0	0	1	
0	0	1	0	$x+y+z'$
0	1	0	1	
0	1	1	0	$x+y'+z'$
1	0	0	1	
1	0	1	0	$x'+y+z'$
1	1	0	1	
1	1	1	1	

Product of sums, POS:  $(x+y+z')(x+y'+z')(x'+y+z')$

## 3.2 Combinational Circuits

### Logic Gates

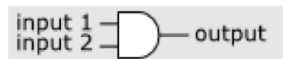
The translation between Boolean algebra and digital logic requires two definitions:

- The correspondence between the values true (encode as high/1) and false (encode as low/0).
- How the operations of conjunction, disjunction and negation are implemented by the circuits.

Gates: the circuits that implement Boolean algebra operations. Given the values at the inputs, the gates almost instantaneously produce the output value according to the operation.

AND Gate (Conjunction)

E.g.  $xy$



Output is 1 if all inputs are 1.

OR Gate (Disjunction)

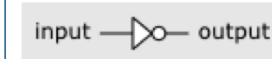
E.g.  $x+y$



Output is 1 if any input is 1.

NOT Gate (Negation)

E.g.  $x'$



Output opposite to input.

NAND Gate (not AND)

E.g.  $(xy)'$



Output is 0 if all inputs are 1. (1 otherwise)

NOR Gate (not OR)

E.g.  $(x+y)'$



Output is 0 if any input is 1. (1 otherwise)

XOR Gate (exclusive OR)

E.g.  $x \oplus y$



Output is 1 if odd number of inputs are 1.

XNOR Gate (not exclusive OR)

E.g.  $(x \oplus y)'$



Output is 1 if even number of inputs are 1.

### Combinational Circuits

Each small gate only implements a simple operation, but the combination of millions of them allow digital circuits to perform sophisticated operations at high speed.

- How: Gates are connected so that the output of one gate becomes the input of another gate.
- Restriction: there cannot be a cycle from the output of a gate to the input of another gate that produced a signal that has been used previously.

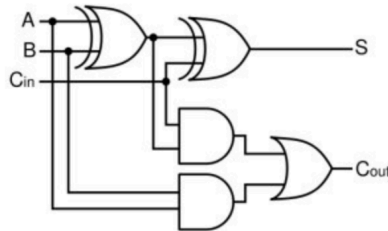
### Example

#### Two Bit Adder with Carry-in and Carry-out

When add two natural numbers, a carry value would be produced that should be added to the next digit.

The truth table and the combinational circuit are shown.

- A, B: input digits
- $C_{in}$ : the carry produced by previous addition
- S: the result of the addition
- $C_{out}$ : the carry produced by this addition



Inputs			Output	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

## 3.3 Translations

### From Combinational Circuits to Boolean Functions

Steps

- 1) Obtain the truth table of the combinational circuit.
- 2) Derive the canonical representation as sum of products or products of sums from the truth table.

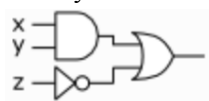
### From Boolean Functions to Combinational Circuits

#### Directly

Step: By observation, using AND, OR and NOT gates to connect the inputs or outputs of some gates.

Example: Boolean expression is  $xy+z'$

Digital Circuit:



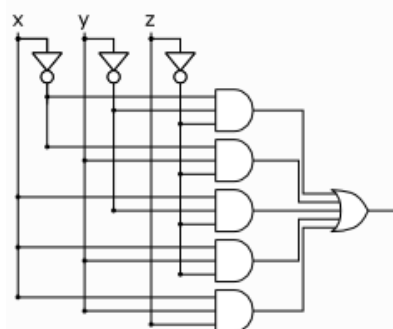
#### Using Sum of Products

Steps

- 1) Derive the Boolean expression as a sum of products from its truth table.
- 2) Each term in the sum of products can be implemented as an AND gate. Using NOT gates for those inputs that are negated. (Number of AND gates = Number of products in the sum).
- 3) Connect the output of all the AND gates as inputs of a single OR gate.

Example: Sum of Products is  $x'y'z' + x'yz' + xy'z' + xyz' + xyz$

Digital Circuit:



The two digital circuits are **equivalent**, but the first one is better as it takes less space and is faster.

# SEQUENTIAL DIGITAL CIRCUITS

Sequential Digital Circuit differs from Combinational Circuits:

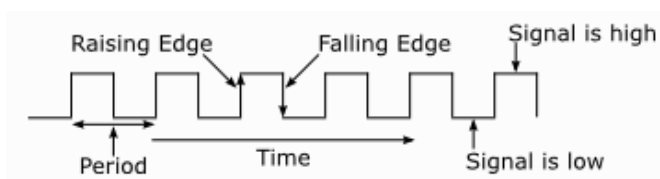
- For combinational circuits, once a combination of inputs is set, the output is 0 or 1 depending on its truth table. The output changes immediately once the inputs change, it does not remember the previous inputs.
- Sequential Digital Circuit can remember the values that occurred at a previous instant.

## 4.1 The Clock

### Definition

Clock: the signal that forces a sequential circuit to behave in one of the two modes:

- Ignore changes in the input values and remembering the last output.
- Pay attention to these inputs and adjust the output accordingly.



Properties

- Overtime, the value of this signal oscillates from high to low level.
- The transitions from high to low levels (called falling edge) and from low to high levels (called raising edge) occur almost instantaneously.
- The period ( $T$ ): time taken by the signal to make a full transition (i.e. from low to high and back to low).
- The frequency ( $f$ ): the speed at which it switches between high and low values.  $f = 1/T$ .

### Sequential Digital Circuits

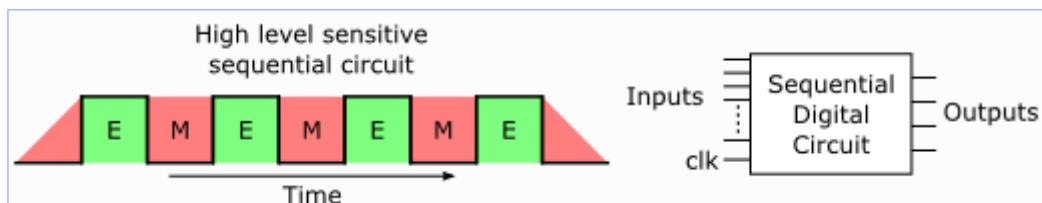
Sequential circuits have clock as an additional input. It is continuously changing between the two behaviours at a pace defined by the clock signal.

Behaviours

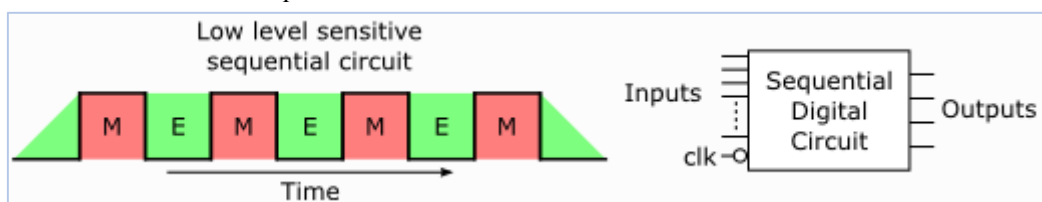
- *Evaluation Time*: at a certain instant marked by the clock, the circuit behaves as a regular combinational circuit and the outputs take the value derived from the inputs.
- *Memory Time*: at any other times, the value of inputs is ignored, and the outputs remain with the values calculated in the previous instance.

### Level-Sensitive Sequential Circuits

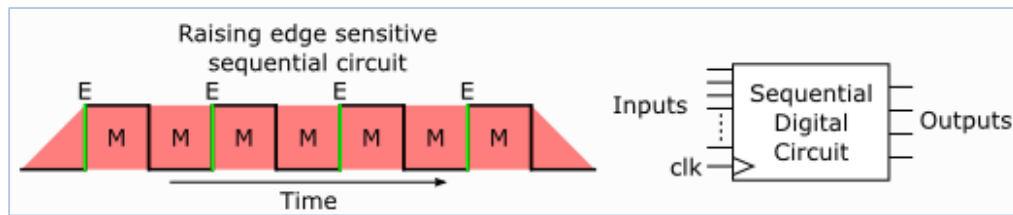
High Level Sensitive Sequential Circuit: **Evaluation time** occurs when the clock is at a **high** level.



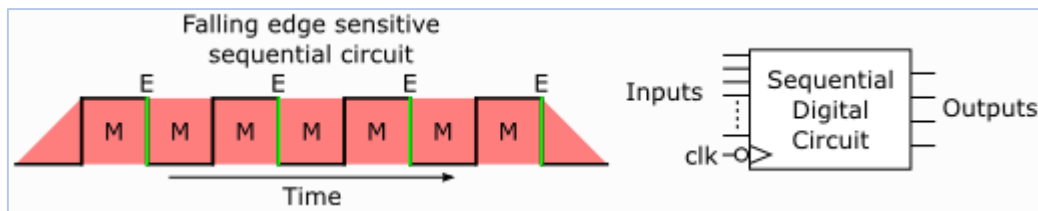
Low Level Sensitive Sequential Circuit: **Evaluation time** occurs when the clock is at a **low** level.



**Raising Edge Sensitive Sequential Circuit.** Evaluation time occurs when the clock makes a transition from **low** to **high** level.



**Falling Edge Sensitive Sequential Circuit.** Evaluation time occurs when the clock makes a transition from **high** to **low** level.

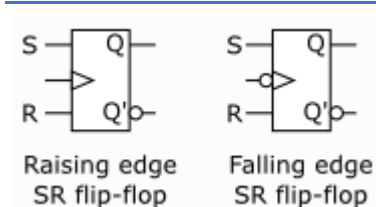


## 4.2 Flip-Flops

A flip-flop is a sequential circuit that is **edge sensitive** and has one or two inputs aside from the clock and two outputs.

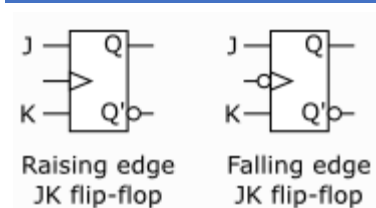
- A flip-flop only makes a transition when allowed by an edge in the clock.
- The values of the two outputs are always **opposite** to each other.
- The output column in the truth table is the future value of the output when the next transition is allowed by the clock, and the output remains constant at any other time.

## SR Flip-Flop



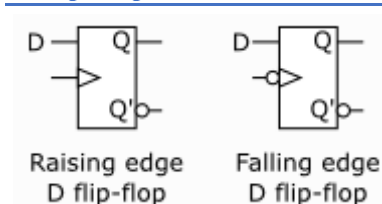
Inputs		Outputs	Explanation
S	R	$Q_{t+1}$	
0	0	$Q_t$	The circuit maintains its output value.
0	1	0	
1	0	1	
1	1	undefined	Undefined state, can't tell if it will be 0 or 1.

## JK Flip-Flop



Inputs		Outputs	Explanation
J	K	$Q_{t+1}$	
0	0	$Q_t$	The circuit maintains its output value.
0	1	0	
1	0	1	
1	1	$Q'_t$	The output becomes the opposite value of the previous output.

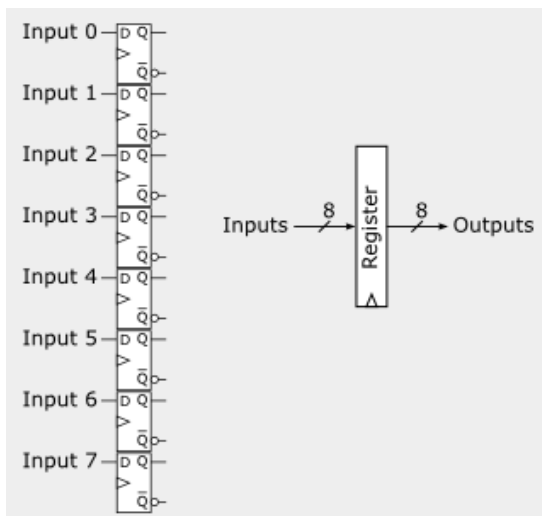
## D Flip-Flop



Input	Output
D	$Q_{t+1}$
0	0
1	1

## Examples of Circuits with Flip-Flop

### Example 1: A Register



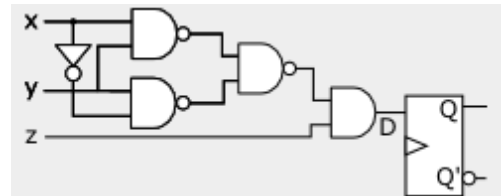
A register is used to store a number of bits given by an equal number of inputs.

Registers are typically implemented by several D flip-flops, the representation of a register storing 8 bits is shown in the diagram.

### Example 2: Boolean Function as Input

The input of a flip-flop can be a Boolean function. The function at the input changes its value at various times, whereas the output of the flip-flop will only change its value at the rising/falling edge of the clock signal.

An example is shown in the diagram, where the input of the D flip-flop is a Boolean function.

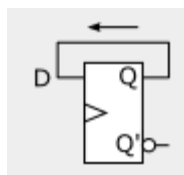


### Example 3: Stable Circuit & Switching Circuit

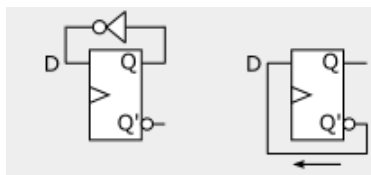
The edge sensitive sequential circuits can use their own output to compute the value at the inputs. However, we need to know the initial value of Q to know future evolution of the circuit.

At the evaluation time, the value of the output is stored in the flip-flop, and will set the input for the next edge to 0 or 1. (Since the edge is passed, change in the input is not registered).

- Stable circuit: if Q has initial value 1 or 0, the circuit will remain with the output at the same value for ever.
- Switching circuit: the circuit will switch the value of its input/output at every rising/falling edge of the clock.



Stable circuit



Switching circuit

### Example 4: Boolean Function with State Inputs

The flip-flop may have a Boolean function as its input, while the Boolean function may have the flip-flop's output as its input.

The truth table of the circuit: For "Input", include inputs of the Boolean function and  $Q_t$  of the flip-flop; For "Output", include  $Q_{t+1}$  of the flip-flop.

## 4.3 Finite State Machines

For Finite State Machines (FSMs), the transition of a sequential circuit is seen as a change of state. The state is defined by the value of the outputs, change of state is defined by the truth table (input variables), and the transition occurs when an edge occurs in the clock.

## Designing FSMs

### Steps

- 1) Identify the different states in the system, and draw each state in a circle with a name.
- 2) Identify the system inputs.
- 3) For each state, consider how the system should evolve for every possible combination of the input values.
- 4) Label each transition connecting two states with the input values required.

Note: The inputs do not necessarily need to be encoded as binary values, but they need to be translated to binary representations when a FSM is implemented as a sequential circuit.

### Example

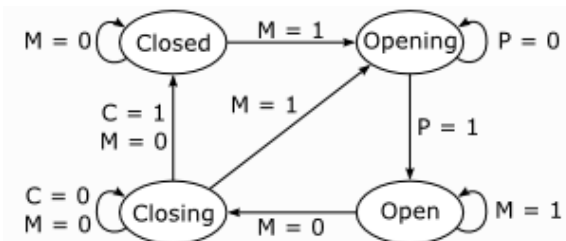
States: closed, closing, opening, open

Inputs: M (motion detector): 0 for no motion, 1 for motion is detected;

P (door motor): 1 for completely open, 0 otherwise;

C (door motor): 1 for completely closed, 0 otherwise.

FSMs:



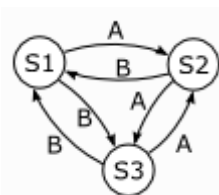
## From FSMs to Sequential Digital Circuit

### Steps

- 1) Identify the possible input values and number of states.
- 2) Encode the input values and the states in binary.
- 3) Create a truth table with "Inputs" columns for each bit of the inputs and current state, and "Output" columns for bits representing the next state.
- 4) Fill in the truth table with all combinations of inputs and states.
- 5) Translate the truth table into a digital circuit using "SOP" or "POS".
- 6) Connect each of the resulting circuits to the input signal of a D flip-flop, and connect the output of the flip-flop to those inputs that represent the current state.

### Example

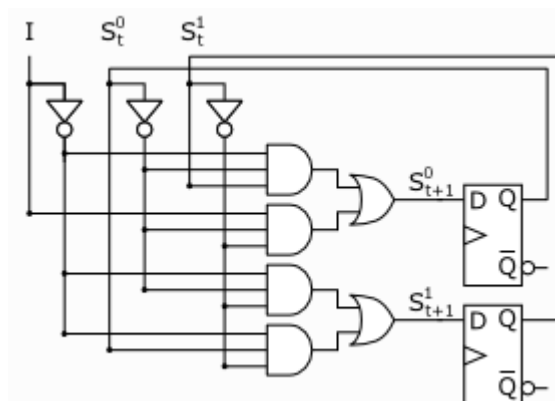
The Finite State Machine is:



We encode  $A = 0$ ,  $B = 1$ ,  $S1 = 00$ ,  $S2 = 01$ ,  $S3 = 10$ . The two bits used to encode the states are  $S^0$  and  $S^1$ .

The truth table and the digital circuit are:

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

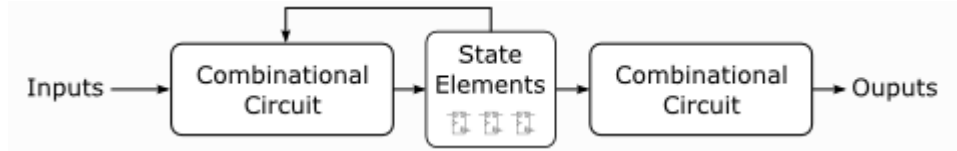


## Output of FSMs

FSMs can be categorized into two types depending on the signals that are considered when producing the outputs.

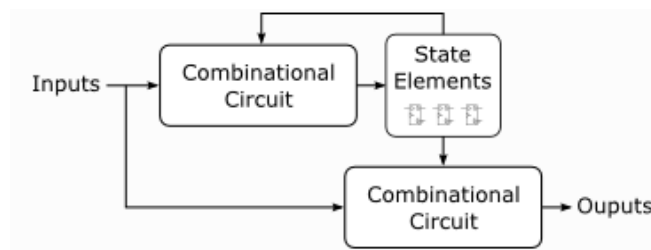
### Moore Machines

Moore Machines are FSMs that produce their outputs only using the signals that encode the state of the machine. The values of the outputs do not depend on the FSM inputs, they only depend on the value of the current state. For example, the output is 00 when the state is S1, the output is 01 when the state is S2, etc.



### Mealy Machines

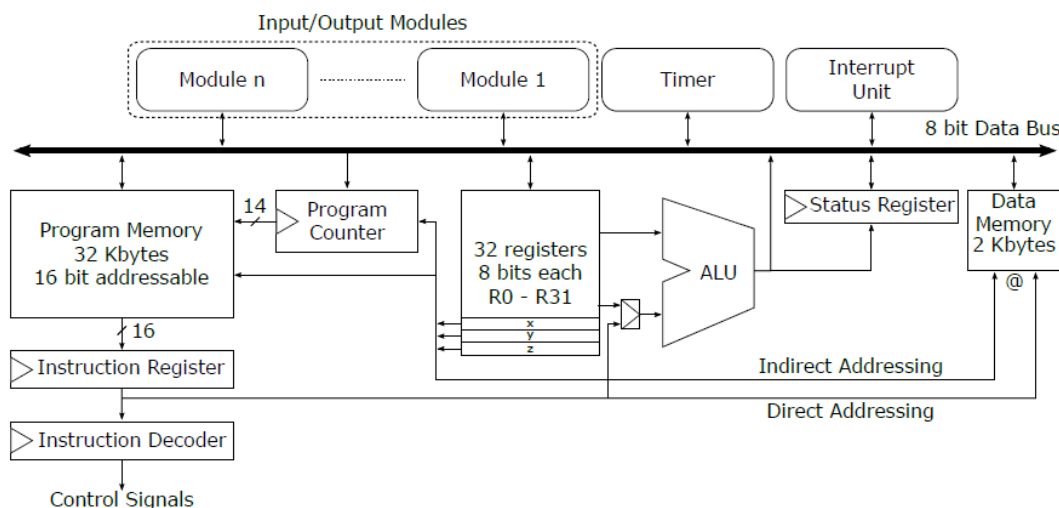
Mealy Machines are FSMs with outputs that may depend on both the current state and the value of the inputs. While being in a state, if the value of the inputs change, the value of the outputs may change as well. For example, the output is 0 when the state is S1 and input I is 0, the output is 1 when the state is S1 but the input I is 1.



# AVR ARCHITECTURE

Architecture: a generic description of a set of microcontrollers that are capable of executing the same machine instructions but that may differ in some characteristics like the size of the memory, speed, etc.

AVR Architecture: a family of microcontrollers that process most of their data in size of 8 bits, and it has a **reduced** set of instructions with **fixed** length format.



Block Diagram of the AVR Architecture



## 5.1 The Execution Environment

The execution environment refers to the elements used to execute instructions: type of data managed; data and status registers; available memory, etc.

Units of information manipulated by a circuit:

Name	Bits	Bytes	Memory Address
Byte	8	1	@
Word	16	2	@ to @+1
Doubleword	32	4	@ to @+3
Quadword	64	8	@ to @+7
Double Quadword	128	16	@ to @+15

### The Data-Path

Data Bus: An 8-bit data bus interconnects all blocks of the AVR Architecture, allowing data to be exchanged among all the modules.

- Modules below the bus: execute the machine language.
- Modules above the bus: execute other tasks including input/ output, interruptions, timer events.

### Program and Data Memories

**Program memory:** stores programs.

- Properties: A flash memory, 16 bits (2 bytes) addressable.
- Size of the memory depends on the microcontroller model. (if size is 32 Kbytes → addresses require 14 bits).
- Purpose: To store the code to execute, which is comprised of instructions of either 16 bits (occupying a single memory cell), or 32 bits (occupying two consecutive cells).

**Data memory:** stores the data that the programs manipulate.

- Properties: A static ram chip, 1 byte addressable.
- Size of the memory depends on the microcontroller model.



Assume the size is 2 Kbytes, the range for address values should be from *0x000* to *0x7FF*.

- However, the chip is designed so that addresses are from *0x000* to *0x8FF*.
- The first 256 positions (*0x000* to *0x100*) are used to access 32 General Purpose Registers and 64 additional input/output registers.

*Memory mapped input/output technique:* Accessing to any of the 256 registers, or accessing data to its corresponding memory address in the data memory are exactly the same operation.

### Program Counter

Program Counter is a very important **register** in any processor, which contains the address in the program memory of the next instruction to be executed.

- The value of the program counter is modified by every instruction.
- Some special instructions modify the register in different ways, such as the branching or jumping instructions.

### General Purpose Registers

Register file: The AVR architecture has 32 8-bit general purpose registers (R0 to R31).

- Size: one byte, as the data memory is 1 byte addressable.
- X (R26: R27), Y (R28: R29), Z (R30: R31): treated as consecutive 16-bit registers which can be used to store the program memory addresses.
- The registers can also be accessed if they were located in the lowest positions of data memory (i.e. addresses 0x000 to 0x01F).

## Arithmetic/Logic Unit (ALU)

ALU is a combinational circuit capable of performing operations.

- Operations: Arithmetic operations (e.g. addition, subtraction); Logical operations (e.g. bit-wise conjunction, 2's complement); Bit operations (e.g. arithmetic and logical shifts, rotations).
- Operands are accessed directly from the register file or the instruction registers.

## Status Register

Status Register is a register where values are stored in some special situations so that other instructions at a later stage can consult and make decisions based on those values.

- Example: Addition with carry.
- The content of the status register should be refreshed every time an operation is executed.
- The size and the type of conditions stored in the status register are dependent on the architecture.

Status Register in AVR architecture has 8 bits:

Bit	Name	Description
0	Carry flag (C)	Indicates if a carry has occurred in the last arithmetic or logic operation.
1	Zero flag (Z)	Indicates if result of the last arithmetic or logic operation has been zero.
2	Negative flag (N)	Indicates if result of the last arithmetic or logic operation has been -ve.
3	2's complement overflow flag (V)	If true, indicates that the last arithmetic or logic operation, if considered over integers encoded in two's complement has produced overflow.
4	Sign flag (S)	The sign indicates the sign of the result of the last arithmetic or logic operation. It is always the exclusive or between the negative and the two's complement overflow flags ( $S=N \oplus V$ ).
5	Half carry flag (H)	Indicates if a carry has occurred at half the operator.
6	Bit copy storage (T)	Source or destination of bit copy operations BST and BLD.
7	Global interrupt enable (I)	If set, the interruptions in the microcontroller are processed. If zero, they are ignored.

## 5.2 The Execution Cycle

The execution cycle of a microprocessor refers to the internal steps to execute an instruction. The number of steps to execute an instruction and their duration vary significantly from processor to processor and are dependent on the architecture.

Two steps to execute instructions in the AVR architecture:

- *Instruction Fetch*: the instruction is obtained from the program memory and stored in the instruction register.
- *Execution*: the instruction is decoded and executed.
- For instructions that require an operation in the ALU and a result written back to register file, the Execution step is divided in three sub-stages: ROF, ALU and RWB.

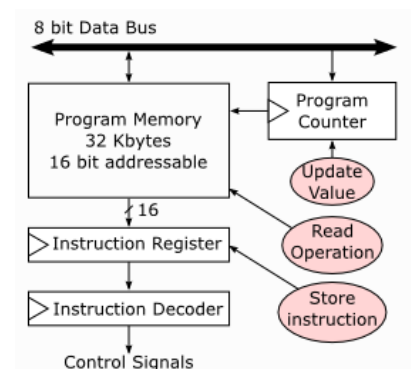
### Instruction Fetch (IF)

IF Steps

- 1) Read the value (i.e. a 16-bits memory address) stored in the program counter by a read operation.
- 2) Store the value in the instruction register.
- 3) At the same time, the program counter is automatically updated to point to the next memory location.

For operations with 32 bits

- An additional value from the program memory is required → a second fetch stage is executed → the 2-bytes value is stored in the instruction register → the value of the program counter is updated.
- This stage takes the same time as the first one.



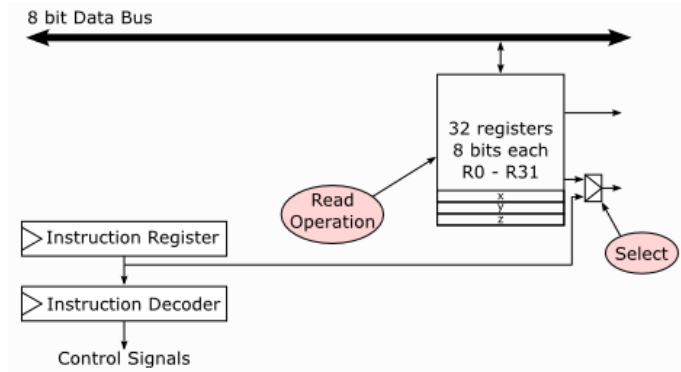
The controller contains a sequential digital circuit:

- **Function:** receives the instruction as inputs, and then over the next clock cycles, generates the appropriate control signals for the rest of the components in the data-path to perform the required operations.
- **Structures:** it captures both the structure of the data-path (because it provides control signals to all their components) and the set of instructions (because it must be capable of interpreting all of them).

### Register Operand Fetch (ROF)

ROF: The operands are obtained from various sources (register file or instruction register).

The picture shows the components that are involved in this stage:



Since most of instructions use the values stored in the register file, the following modes of operation with respect to inputs and outputs are designed:

- 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 Special Cases:

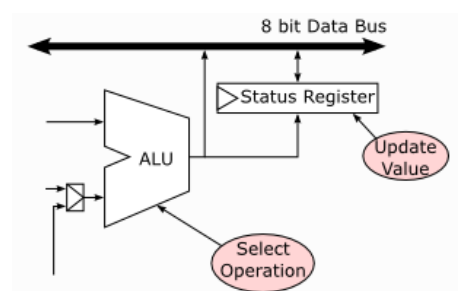
- For instructions containing one operand that is part of the instruction itself, a pre-defined subset of the instruction bits is extracted and used as operand.
- For instructions that read or write data from the Data Memory, the operands required are used to calculate the address of the memory from where the data will be read or written.

### ALU Execution (ALU)

Note: This stage is only present in the instructions that require an operation to be performed by the ALU.

ALU Steps

- 1) The control signals select the appropriate operation, and ALU produces the result at the output.
- 2) The status register is updated to reflect the conditions of the result, the change of bits is according to the table shown below.



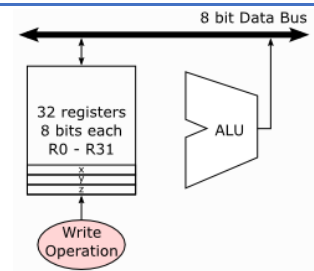
Bit	Conditions of Change of Bits
Carry Flag	If a carry has occurred in the operation.
Zero Flag	If the result is zero.
Negative Flag	If the result has the sign bit to one.
Two's Complement Overflow Flag	If the overflow condition for two's complement is satisfied, that is if the carry into the most significant bits is different to the carry derived from the most significant bit.
Sign Bit	The exclusive or between the Negative Flag and the Two's complement Flag.
Half Carry Flag	If there is a carry out of the third bit of the result.

## Register Write Back (RWB)

RWB: present when the result produced in the ALU stage needs to be stored in the register file.

### RWB Steps

- 1) The ALU writes the result in the data bus.
- 2) At the same time, the register file reads this result from the bus, and writes into the destination selected.



## Pipelined Execution

### 2-Stage Pipelining Technique

- **Purpose:** To increase the overall performance of the microcontroller.
- **How:** To execute the two main stages of two instructions in parallel to increase speed. The execution stage of an instruction is done in parallel with the fetch of the next instruction.
- **Limitations:** Execution of instructions in parallel increases the overall circuit design, as the control signals need to be generated in parallel, and some instructions may break the normal sequence of instructions (jumps).

## 5.3 The Stack

The stack is an area in the data memory where data can be read and written with special instructions. It is used to store temporary data during short time intervals.

**Stack Overflow:** The space reserved for stack is usually small. Stack overflow occurs when an unusually large number of data is pushed to the stack.

### The Stack Pointer

The stack pointer: the register that store the address of the **top** of the stack. It is part of the generic Input/ Output register in the AVR architecture.

The 16-bit address of the top of the stack is stored in two registers:

- Register SPH (@0x3E): Store the most significant 8 bits of the address of the top of the stack.
- Register SPL (@0x3D): Store the least significant 8 bits of the address of the top of the stack.

### Stack Initialization

Stack Initialization Steps:

- 1) Reserve memory specially for the stack.
- 2) Set the correct value in the stack pointer.

In the AVR architecture the stack pointer is typically initialized at the last position in data memory.

### Stack Instructions

Stack Instructions: pushing data from a location to the stack; popping data from the stack to a location.

Notations Used:

- @top: address of the top of the stack.
- $r_n$ : any of the general purpose registers.

	Push instructions	Pop instructions
Syntax	<b>push</b> $r_n$	<b>pop</b> $r_n$
Effect	Store the value of the register $r_n$ on top of the stack.	Store the value at the top of the stack in the register $r_n$ and adjust @top.
Steps	<ol style="list-style-type: none"><li>1) The value of @top is decremented by one (<math>@top = @top - 1</math>).</li><li>2) The register <math>r_n</math> is written in the memory location @top. The data previously stored in this memory location is lost.</li></ol>	<ol style="list-style-type: none"><li>1) The value pointed by @top is read from the data memory and written in the register.</li><li>2) The value of @top is incremented by one (<math>@top = @top + 1</math>).</li></ol>

### Other Instructions Using the Stack

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.

## AVR INSTRUCTION SET

Instruction Set Architecture (ISA): the set of instructions that a microprocessor can execute. It describes all the details about the execution of each instruction and the effects in the components of the microprocessor.

### 6.1 Types of Instruction Sets

For a microprocessor, the type of instructions it supports is influenced by:

- The processor architecture.
- The complexity of the implementation: the higher the complexity of the tasks, the higher the complexity of the digital circuit design → large no. of gates → larger physical space and more power consumption.

Microprocessors can be divided into two types:

- According to the instructions encoding format: Fixed length format, and Variable length format. (For more details, refer to Section 2.2 Machine Instruction).
- According to the complexity of machine languages: CISC and RISC.

#### Complex Instruction Set Computers (CISC)

CISC: the processors that provide a rich and complex set of instructions.

- Property: complex set of instructions, take longer to execute, but shorter instruction sequences.
- Examples: Intel's x86, IA-32, IA-64, IBM's System 360.

CISC Instructions: typically use several operands and may require multiple memory accesses, the memory address may be obtained by certain arithmetic operations.

- Example: ADD \$4, 14(%eax, %ebx, 8)  
Explanation: Add the value 4 to the data stored in the memory address, obtained by adding the value 14, the content of register %eax, and the content of register %ebx multiplied by 8.

#### Reduced Instruction Set Computer (RISC)

RISC: the processors that provide a reduce and simple set of instructions.

- Property: simpler set of instructions, faster to execute and decode due to the simpler operation, but longer instruction sequences.
- Examples: MIPS, ARM, SPARC, PowerPC, AVR.

RISC Instructions

- Example: ADD R5, R7  
Explanation: Add the contents in register 5 and register 7, store the result in register 5.

#### Comparison

Comparison between CISC and RISC:

- Cannot be done simply in terms of the number of instructions executed per unit of time.
- A better way is to execute the same complex task in CISC and RISC processors and measure the amount of time they take to finish the task.

## 6.2 Instruction Format

**Operation Code:** encode the types of operation, occupy 4 to 8 bits of the operation.

**Rd:** A register 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.

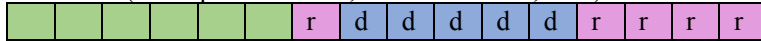
**K:** A constant value. (i.e. immediate operands).

**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.

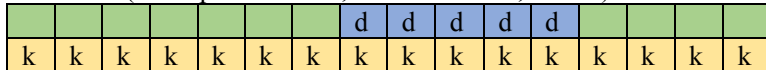
**Format 1** (Example: ADD Rd, Rr → ADD R0, R31)



**Format 2** (Example: CPI Rd, K → CPI R16, 255)



**Format 3** (Example: LDS Rd, k → LDS R12, 12565)



## 6.3 The Assembly Language

**Assembly Language:** a direct alphanumeric representation of instructions executed by a microprocessor as part of its machine language. The assembly code is highly dependent on the architecture.

AVR Assembly Language Rules

- Syntax: start with the instruction mnemonic, then the destination operand, the source operand.
- Case insensitive.
- Registers: R0 to R31/ r0 to r31.
- Constants: simply represented as numbers.
- Memory Address: referred by its label, which must be previously defined in the data section.

### Instructions to Transfer Data

Move data from one location to another location, including memory or general purpose registers.

Syntax	Description	Operation	Restriction
<b>MOV</b> Rd, Rr	Copy value of Rr into Rd	$Rd \leftarrow Rr$	No
<b>LDI</b> Rd, K	Load immediate K into Rd	$Rd \leftarrow K$	$16 \leq d \leq 31, 0 \leq K \leq 255$
<b>LD</b> Rd, X <b>LD</b> Rd, X+ <b>LD</b> Rd, -X	Load one byte from the memory address stored in the register X into register Rd	$Rd \leftarrow (X)$ $Rd \leftarrow (X), X \leftarrow X+1$ $X \leftarrow X-1, Rd \leftarrow (X)$	Source Operand can be X, Y, Z
<b>LDD</b> Rd, Y+q	Load one byte from the memory address into Rd	$Rd \leftarrow (Y+q)$	Source Operand: <b>Y, Z only</b> $0 \leq q \leq 63$
<b>LDS</b> Rd, k	Load one byte from the data space directly to Rd	$Rd \leftarrow (k)$	$0 \leq k \leq 65535$
<b>ST</b> X, Rr <b>ST</b> X+, Rr <b>ST</b> -X, Rr	Store the value of Rr into the address pointed by X	$(X) \leftarrow Rr$ $(X) \leftarrow Rr, X \leftarrow X+1$ $X \leftarrow X-1, (X) \leftarrow Rr$	Destination Operand can be X, Y, Z
<b>STD</b> Y+q, Rr	Store value of Rr into the indirect memory address	$(Y+q) \leftarrow Rr$	Source Operand: <b>Y, Z only</b> $0 \leq q \leq 63$
<b>STS</b> k, Rr	Store value of Rr into the data memory at address k	$(k) \leftarrow Rr$	$0 \leq k \leq 65535$
<b>PUSH</b> Rr	Store value of Rr into the stack	$SP \leftarrow SP-1, (SP) \leftarrow Rr$	No
<b>POP</b> Rd	Load value at the top of the stack into Rd	$Rd \leftarrow (SP), SP \leftarrow SP+1$	No

## Arithmetic Instructions

Arithmetic Instructions include addition, subtraction, increment, decrement, multiplication and negation.

Syntax	Description	Operation	Restriction
<b>ADD</b> Rd, Rr	Add two registers, store result in Rd	$Rd \leftarrow Rd + Rr$	No
<b>SUB</b> Rd, Rr	Subtract two registers, leave result in Rd	$Rd \leftarrow Rd - Rr$	No
<b>SUBI</b> Rd, K	Subtract constant K from Rd, leave result in Rd	$Rd \leftarrow Rd - K$	$16 \leq d \leq 31$ $0 \leq K \leq 255$
<b>INC</b> Rd	Increment the value in Rd by 1	$Rd \leftarrow Rd + 1$	No
<b>DEC</b> Rd	Decrement the value in Rd by 1	$Rd \leftarrow Rd - 1$	No
<b>NEG</b> Rd	Change the sign of the value in Rd	$Rd \leftarrow 0 - Rd$	No
<b>MUL</b> Rd, Rr	Multiplication of two unsigned 8-bit values, result stored in R1 (high byte) and R0 (low byte)	$R1:R0 \leftarrow Rd \times Rr$	No


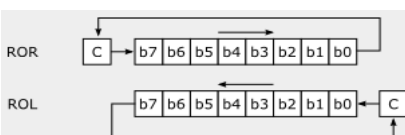
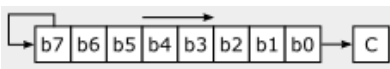
## Logic Instructions

Bit-wise Logic Operations pair the 8 bits of the operands in the same position and perform the basic operation.

Syntax	Description	Operation
<b>AND</b> Rd, Rr	Perform logical AND between values of Rd, Rr	$Rd \leftarrow Rd \bullet Rr$
<b>OR</b> Rd, Rr	Perform logical OR between values of Rd, Rr	$Rd \leftarrow Rd \vee Rr$
<b>EOR</b> Rd, Rr	Perform logical exclusive OR between values of Rd, Rr	$Rd \leftarrow Rd \oplus Rr$

## Shift and Rotate Instructions

Used to access the bits inside a byte independently.

Syntax	Description	Operation
<b>LSR</b> Rd <b>LSL</b> Rd	Shift all bits in Rd by one place to the right/ left. <ul style="list-style-type: none"><li>LSR: clear b7 of Rd, load b0 with the value of C flag of the status register. (/2)</li><li>LSL: clear b0 of Rd, load b7 into the C flag of the status register. (*2)</li></ul>	
<b>ROR</b> Rd <b>ROL</b> Rd	Rotate right/ left through Carry, shifting the carry bit C into bit 7/ bit 0.	
<b>ASR</b> Rd	Shifts all bits in Rd one place to the right, bit 7 held constant, bit 0 is loaded into the C flag. (/2 maintaining its sign)	

## Compare Instructions

Syntax	Description	Operation	Restriction
<b>CP</b> Rd, Rr	Comparison between two registers	$Rd - Rr$	No
<b>CPI</b> Rd, K	Comparison between constant K and Rd	$Rd - K$	$16 \leq d \leq 31, 0 \leq K \leq 255$

## Jump and Branch Instructions

Used to change the sequence of instruction execution, always used in conditionals and loops.

Jump: always jump to a new location. Branch: jump only if a specific condition is met.

Syntax	Description	Operation	Restriction
<b>JMP</b> k	Jumps to @k of program memory	$PC \leftarrow k$	$0 \leq k < 2^{22}$
<b>BREQ</b> k <b>BRNE</b> k	Branch if equal Branch if different	If $(Z = 1)$ , $PC \leftarrow PC + k + 1$ , else $PC \leftarrow PC + 1$ . If $(Z = 0)$ , $PC \leftarrow PC + k + 1$ , else $PC \leftarrow PC + 1$ .	$-64 \leq k \leq 63$
<b>BRSH</b> k <b>BRLO</b> k	Branch if equal or higher Branch if lower (unsigned)	If $(C = 0)$ , $PC \leftarrow PC + k + 1$ , else $PC \leftarrow PC + 1$ . If $(C = 1)$ , $PC \leftarrow PC + k + 1$ , else $PC \leftarrow PC + 1$ .	$-64 \leq k \leq 63$
<b>BRGE</b> k <b>BRLT</b> k	Branch if equal or greater Branch if lower (signed)	If $(S = 0)$ , $PC \leftarrow PC + k + 1$ , else $PC \leftarrow PC + 1$ . If $(S = 1)$ , $PC \leftarrow PC + k + 1$ , else $PC \leftarrow PC + 1$ .	$-64 \leq k \leq 63$



Compare instructions and Branch instructions are usually used together.

- Syntax: CPI A, B  
BR[CONDITION] destination
- The code will branch to the “destination” if the condition “A CONDITION B” is satisfied.
- Example BRGE: the process will branch if A is greater or equal to B.

### Subroutine Call and Return

Syntax	Description	Operation	Restriction
<b>CALL</b> k	Call a subroutine. Return address is stored in the stack. The stack pointer (SP) is decremented to point to the return address.	$PC \leftarrow k$ , $SP \leftarrow SP-2$ , $STACK \leftarrow PC+2$	$0 \leq k < 2^{22}$
<b>RET</b>	Return to the instruction following the instruction CALL last invoked	$PC \leftarrow STACK$ , $SP \leftarrow SP+2$	No

### Input and Output Instructions

Transfer of data from/ to the input/ output ports.

Syntax	Description	Operation	Restriction
<b>IN</b> Rd, A	Read the value of the input port in location A, store in Rd	$Rd \leftarrow I/O[A]$	A: address of an input port
<b>OUT</b> A, Rd	Store/ write value of Rd in output location A.	$I/O[A] \leftarrow Rd$	A: address of an output port

## AVR ASSEMBLY PROGRAMS

Assembler: a program that given a set of plain text files containing assembly code, produces a binary file containing the binary encoding of all the instructions.

Compiler: a program that translates a program from one programming language to another.

### 7.1 Assembly Program

An assembly program is written in files with extension “.s”. The syntax of assembly code introduced in following sections is accepted by the assembler “avr-gcc”.

Every assembly program must follow the basic structure shown on the right.

Key Words

- **.section .data**: a mark to tell the assembler that the following lines are data definitions.
- **.section .text**: a mark to tell the assembler that the followings are assembly instructions.
- **.global main**: declare the word “main” as a global symbol, thus the location in the code with the label “main” can be accessed from outside of the file.

```
.section .data
.section .text
.global main
main:
    ret
.end
```

### Data Definition in Data Section

All the data defined in the data section is stored in consecutive memory locations. The address can be accessed by defining a “label” at the beginning of the line.

Difficulty to manipulate data: there is no information stored to remember the type of data or its structure.

Definition	Example	Note
Byte, Integer	label: <b>.byte</b> 23, 0xFF, ... , 'A', -12	Range for each value: -256 to 255 (1 byte).
String	label: <b>.ascii</b> “Str A”, ... , “Str B”	Each character is encoded with 1 byte.
String	label: <b>.asciz</b> “Str A”, ... , “Str B” label: <b>.string</b> “Str A”, ... , “Str B”	Each character is encoded with 1 byte. And each string is encoded with an extra byte <b>0x00</b> as last character.
Empty Space	label: <b>.space</b> 4, 0xFF	Reserve 4 memory spaces, initialized with 0xFF. (If the second number is omitted, initialize with 0).



## Labels

Label: a string written at the very beginning of a line in the program and are followed by a colon.

- The label name is used as a reference to the memory address of the location in the program.
- Labels must be unique throughout the entire file.

Labels can be used for both data and code addresses.

Sections	Function	Example	Explanation
data	refer to the data	<i>label</i> : .byte 23, 0xFF	<i>label</i> represents the address of the first byte of these values.
text	refer to a data	LDS R12, <i>label</i>	load the value stored at <i>label</i> address into R12.
text	used to obtain any relative address	LDI R28, lo8( <i>label</i> ) LDI R29, hi8( <i>label</i> )	The <i>label</i> address is loaded into the register Y. <ul style="list-style-type: none"><li>• lo8(<i>label</i>)/ hi8(<i>label</i>) returns the 8 least/ most significant bits of the address assigned to <i>label</i>.</li><li>• Then, we can use <i>Y+q</i> to obtain other address, where <i>q</i> is a constant.</li><li>• Cannot use <i>label+q</i> directly to access addresses.</li></ul>

## Stack Restriction

The processor may use the stack to store additional data, so there are restrictions on stack:

- The top of the stack must be exactly the same before the execution of the first instruction of a routine, and before the execution of the last instruction (i.e. ret). → The stack's content must be restored to its initial state before the end of the program.

## Register Restriction

Restrictions on Register

- R0: used as scratch register, and need not to be saved nor restored.
- R1: must be kept always at value zero. (need save and restore)
- R2 to R17, Y (i.e. R28 and R29): must be saved and restored at the beginning and end of a subroutine. Untouched if subroutine is called.
- R18 to R27, Z (i.e. R30 and R31): may not be saved before its use, but may be overwritten when a function is called from the code.
- If a function returns an integer, this must be placed in register R25: R24 at the end of the subroutine.

If the registers are stored in the stack, save and restore can be done using *push* and *pop* instructions. Note that the order of the *push* instructions at the beginning of the subroutine is symmetric to the order of the *pop* instructions before the end of the subroutine.

## 7.2 Guidelines for Assembly Programming

When the assembly program does not do the expected work, we need to review the instructions line by line to find out the anomaly.

### Recommendations

Things to Check:

- The values in registers R1 to R17, R28 and R29 must be the same to the values present before executing the program/ subroutine.
- Avoid unnecessary operations. E.g. do not save and restore registers that are not needed or unimportant content. Move the data to the right location to avoid unnecessary data movement operations.
- There are always several ways to program a task. Try to choose the one with less number of instructions or that you think it will execute faster. Memory accesses typically slow down program execution.
- Write legible code. Write comments before blocks of instructions, indicating the function and the purpose of the instructions. Do not write trivial comments about a single instruction, but instead, high level comments about code blocks.

## Example

```
;; Data definitions
.section .data
n1:    .byte 12
n2:    .byte 34
n3:    .byte 21
result: .space 1

;; Code definition
.section .text
.global main

main:
    PUSH R17
    LDI R26, lo8(n1)      ; Loading the address of n1 in X
    LDI R27, hi8(n1)
    LD R24, X+            ; Load n1 in R24 and increase address
    LD R17, X+            ; Load n2 in R17 and increase address
    ADD R24, R17          ; R24 = n1 + n2
    LD R17, X+            ; Load n3 in R17 and increase address
    ADD R24, R17          ; R24 = n1 + n2 + n3
    ST X, R24             ; Store the result in X
    CLR R25               ; So that R25: R24 has the result in 16 bits.
    POP R17
    ret
.end
```

Five registers are used in the example:

- R17: to temporarily store the values loaded from memory.
- X (R27: R26): to store the address of the data being accessed.
- R24: to accumulate the result of additions.
- R25: to return the result as R25: R24.

# ADDRESSING MODES

After an instruction is loaded from memory into the instruction register, the microprocessor recognizes the type of instruction and then obtains the required operands.

This chapter will introduce how to obtain operands. Operands are stored in:

- General purpose registers → easy to access
- Instruction itself → easy to access
- Memory → need additional operations to manipulate the address

## 8.1 Notations

- $@_e$ : the effective address of an operand, which is the place where the operand is stored.
- $inst$ : the instruction used to calculate  $@_e$ .
- $@_{inst}$ : the address where the instruction is stored.
- $inst_{f1}$  to  $inst_{fk}$ : instruction's fields.
- $R_{fi}$ : the register will be denoted by  $R_{fi}$  if it is encoded by the field  $fi$ .
- $DMEM[a]$ : the content stored in data memory in position  $a$ .
- $PMEM[a]$ : the content stored in the program memory in position  $a$ .
- $R \leftarrow v$ : loading the value  $v$  into the register  $R$ .

## 8.2 Addressing Modes

Operands in one machine instruction can be divided into four categories: constants, registers, memory addresses, implicit operands. The first three types of operands are obtained using addressing modes.

### Register Direct

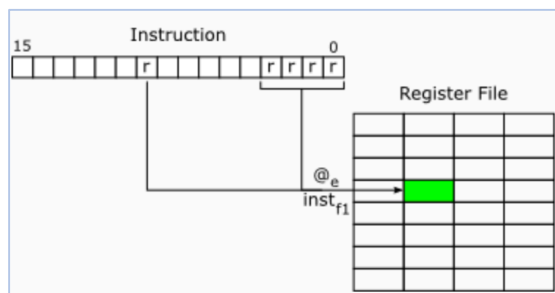
Register Direct addressing mode is used to obtain operands that are stored in a general purpose register.

Example Instruction: ADD Rd, Rr

Instruction Encoding: 0000 11rd dddd rrrr

Equations to capture Register Direct:

- $@_e = \text{inst}_{f1}$  (a register address inside the register file)
- $\text{operand} = R_{f1}$



### Immediate

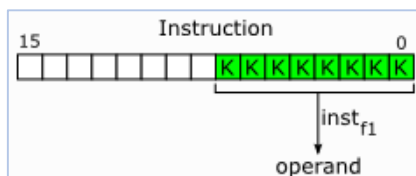
Immediate addressing mode is used to specify the constant as part of the instruction.

Example Instruction: LDI R18, 15;

Instruction Encoding: 1110 kkkk dddd kkkk

Equations to capture Immediate:

- $@_e = \text{PC}$  (the program counter contains the address of the instruction in the program memory).
- $\text{operand} = \text{inst}_{f1}$  (the value of the instruction field)



Restrictions:

- Only be used when loading a value into a register with the instruction LDI.
- The constant is encoded by 8 bits  $\rightarrow$  value between 0x00 to 0xFF.
- LDI only allows 4 bits to encode the register  $\rightarrow$  only R16 to R31 can be used as destination register.

### Data Direct

Data Direct addressing mode is used to obtain operands which are stored in the data memory, with the memory address as part of the instruction.

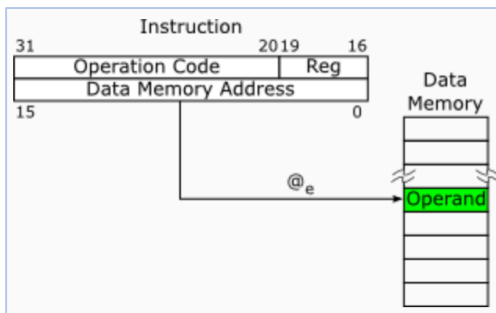
Example Instruction: LDS R12, label

Data Direct has a longer execution cycle because:

- Requires three memory access. Two accesses to the program memory to get the instruction, one access to the data memory to read/ write the operand.
- The program counter will be updated twice, once to access the address stored in the program memory, once to leave it pointing to the next instruction.

Equations to capture Data Direct:

- $@_e = \text{PMEM} [@_{inst} + 1]$  //note that the program memory has cell size of 2 bytes
- $\text{operand} = \text{DMEM} [\text{PMEM} [@_{inst} + 1]]$



## Data Indirect

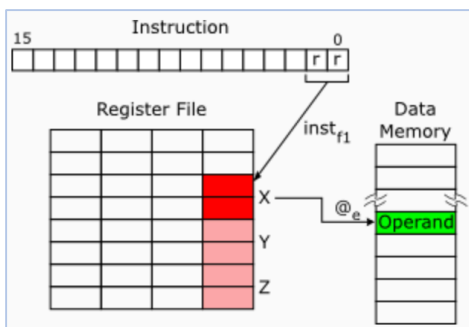
Data Indirect addressing mode is used to obtain an operand stored in the data memory, with the memory address stored in one of the three 16 bit registers X, Y or Z.

Example Instruction: LD R12, X

Restriction: The register can only be X, Y or Z.

Equations to capture Data Indirect:

- $@_e = R_{f1}$  (the content of the register encoded in the instruction field)
- $\text{operand} = \text{DMEM} [R_{f1}]$



Two memory accesses: The instruction fetch to access the instruction; Access the operand from data memory. Data Indirect is useful when accessing a memory address several times, or when the address of a data is the result of a previous arithmetic operation.

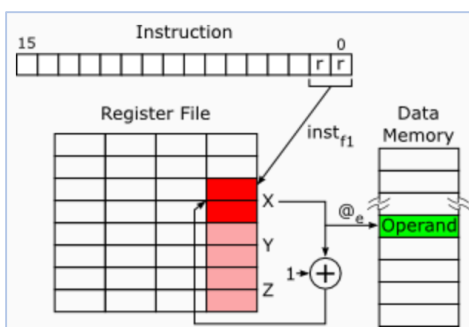
## Data Indirect with Post-Increment

Similar to Data Indirect, but the register containing the address is automatically **increased** by one and updated with the new value. The increment is performed **after** the value has been used.

Example: LD R12, X+

Equations to capture Data Indirect with Post-Increment:

- $@_e = R_{f1}$  (the content of the register encoded in the instruction field)
- $\text{operand} = \text{DMEM} [R_{f1}]$
- $R_{f1} \leftarrow R_{f1} + 1$



This addressing mode is very useful to access data that is stored in consecutive data memory locations and the address of its first byte is already pre-loaded in X/ Y/ Z.

Application: The instruction POP uses this addressing mode.

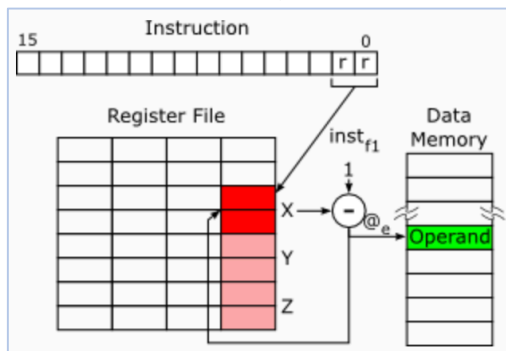
### Data Indirect with Pre-Decrement

Similar to Data Indirect, but the register containing the address is automatically **decreased** by one and updated with the new value. The decrement is performed **before** the value is used.

Example: LD R12, -X

Equations to capture Data Indirect with Pre-Increment:

- $R_{f1} \leftarrow R_{f1} - 1$
- $@_e = R_{f1}$  (the content of the register encoded in the instruction field)
- operand = DMEM [ $R_{f1}$ ]



This addressing mode is useful when the data are accessed in decreasing memory locations.

Application: The instruction PUSH uses this addressing mode.

### Data Indirect with Displacement

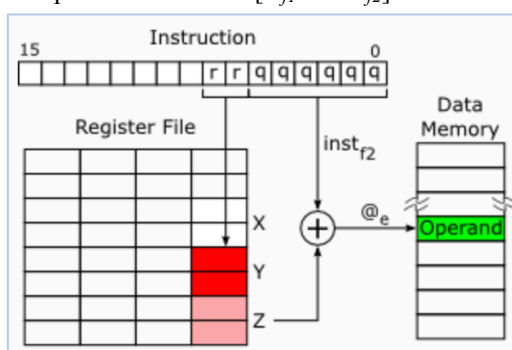
Data Indirect with Displacement is used to access the fields of a complex data structure.

Example: LDD R12, Y+23

Restriction: Only registers Y or Z can be used. The displacement d must satisfy  $0 \leq d \leq 63$ .

Equations to capture Data Indirect with Displacement:

- $@_e = R_{f1} + inst_{r2}$  (adding two instruction fields)
- operand = DMEM [ $R_{f1} + inst_{r2}$ ]



How: Knowing the number and types of fields, we will know their size in bytes, thus displacement to be added to the first address to access each of them.

Two memory accesses: The instruction fetch; access the operand.

# HIGH LEVEL PROGRAMMING

## 9.1 High Level Programming

### Translation of if-then-else Statements

Programming Language	Assembly Code	
if (Boolean expression) { Statement A; } else { Statement B; }	... ; Evaluate the expression. ... ; Store the result in R18 CPI R18, 0 ; Compare result with False (encoding as 0) BREQ block B ; If the result is false, jump to block B (Statement A) ; Instructions translating Statement A JMP end_if ; Jump to the end block B: (Statement B) ; Instructions translating Statement B end if:	
When Boolean expressions contain two or more expressions	Boolean Expression: e1 && e2 e1 false Jump to block B e2 false Jump to block B Statement A	Boolean Expression: e1    e2 e1 true Jump to block A e2 false Jump to block B

### Translation of Switch

Programming Language	Assembly Code	
switch (Expression) { case Value1: Statement1 break; case Value2: Statement2 break; default: Default Statement }	... ; Evaluate the expression. ... ; Store the result in R18 CPI R18, Value1 ; Compare result with Value1 BREQ case1 ; If equal, jump to case1 CPI R18, Value2 ; Compare result with Value2 BREQ case2 ; If equal, jump to case2 JMP default ; Only if default is present case1: (Statement 1) ; Instructions translating Statement 1 JMP done ; Only if break is present case2: (Statement 2) ; Instructions translating Statement 2 JMP done ; Only if break present default: (Default Statement) ; Instructions translating default statement done:	

\*If no "break" at the end, the next block is executed independently of the value of the expression.

### Translation of While Loops

Programming Language	Assembly Code
While (Boolean expression) { Statements; }	loop_start: ... ; Evaluate the expression ... ; Store the result in R18 CPI R18, 0 ; Compare result with 0 (False) BREQ done ; If result is False, jump out the loop (Statements) ; Instructions translating Statements ... ; Use INC and STS if update is present JMP loop_start ; Go back to evaluation code done:

When update is present: see the example in later section for more details.

## Translation of For Loops

Programming Language	Assembly Code
for (Initialization; Boolean condition; Update) { Statements; }	... ; Initialization loop_start: ... ; Evaluate the expression. ... ; Store the result in R18 CPI R18, 0 ; Compare result with 0 (False) BREQ done ; If result is False, jump out the loop (Statements) ; Instructions translating the Statement ... ; Use INC and STS if update is present JMP loop_start ; Go back to evaluation code done:

## Example

	If-else	For loop
Programming Language	if ((x <= 3) && (i == (j+1))) { Statement A; } else { Statement B; } 	for (i = 0; (i < size) && (j != 0); i++) { Statements; }
Assembly Code	LDS R18, x LDI R19, 3 CP R19, R18 BRLT blockB LDS R18, j INC R18 LDS R19, i CP R18, R19 BRNE blockB (Statement A) JMP end_if blockB: (Statement B) end_if:	LDS R18, i CLR R18 STS i, R18 loop_start: LDS R19, size CP R18, R19 BRGE done LDS R20, j CPI R20, 0 BREQ done (Statements) INC R18 STS i, R18 JMP loop_start done:

## 9.2 Subroutine Execution

A subroutine is a set of instructions that operate over a set of values placed in a specific location, and that can be invoked from any other location in the code.

Advantages:

- Avoid redundant code. Place the operations in a subroutine which may be performed over different data in several instances of the program execution.
- Promote the division of tasks, as complex tasks can be decomposed in simpler steps.
- Promote code encapsulation → promote code reuse.

Disadvantage: need to establish a calling convention, a protocol starting where and how to pass the parameters and return the result. There will be additional instructions to perform these steps.

### Call and Return from Subroutines

Call a Subroutine: Execute the instruction “CALL *subroutine*” → The microprocessor fetches the instruction in the memory position corresponding to *subroutine*. The program counter points to the instruction following CALL, the value is stored as the return address → Execute instructions in the subroutine → RET → The microprocessor returns to the point in which subroutine was called and fetches the instruction following the CALL.

For Nested Invocations:

- The CALL pushes its next instruction to the top of the stack, the return addresses will be stored in the stack in the order of invocation. RET instructions will simply obtain the return address available at the top of the stack.
- Restriction: when a subroutine executes its last instruction, the stack pointer must be pointing to the same memory address when the subroutine started.

## Parameters and Result

### Definitions

- Parameters: copies of values that are made available to the subroutine by the caller, and are disposed of after the subroutine has finished.
- Returned result: a value produced by subroutine and copied in a specific location for the caller to use it.

The following content introduces three different calling conventions where parameters and results are stored in registers, memory or the stack.

### Registers

This calling convention defines the registers and their order which are used by the calling program to place the parameters, and the register which are used by the invoked subroutine to return the result.

- Disadvantage: limited number of registers to store the parameters and results.
- Advantage: efficiency. Temporary values are stored in the general purpose registers → simple operations to use or collect the value in registers.

Example: Parameters can be placed in registers from R25 to R8, result can be placed in register R25: R24.

Calling Program	Invoked Subroutine
<pre>... LDS R25, x      ; First parameter LDS R24, y      ; Second parameter CALL subroutine MOV Y+, R25 MOV Y+, R24     ; Store result in Y</pre>	<pre>subroutine: MOV R5, R25     ; first parameter MOV R4, R24     ; second parameter ... LDI R25, 0xFF   ; result LDI R24, 0xFF   ; result RET</pre>

### Memory

This calling convention defines a memory area known to both the calling program and the invoked subroutine, which will store the copy of parameters and the space where the result will be placed.

- Advantage: the amount of space allocated to store the parameters is bigger.
- Disadvantage: need define the memory space to store parameters and result, additional instructions required to transfer the parameters from and to memory.

Limitations:

- Undetermined number of parameters → The memory area to store parameters cannot be defined.
- In the case of recursive functions, functions call themselves → need to store additional parameters and results for new invocation while keeping the current copy of parameters.

Example:

Data Section	Calling Program	Invoked Subroutine
<pre>p1: .space 1, 0 p2: .space 1, 0 result: .space 2, 0</pre>	<pre>... STS p1, R25     ; First param STS p2, R24     ; Second param CALL subroutine LDI R28, lo8(result) LDI R29, hi8(result) LD R24, Y LDD R25, Y+1    ; Get result</pre>	<pre>subroutine: ... LDS R19, p1     ; first param LDS R18, p2     ; second param ... LDI R28, lo8(result) LDI R29, hi8(result) ST Y, R24       ; result STD Y+1, R25    ; result RET</pre>



### The Stack

In this calling convention, the calling program will push to the stack all the parameters and some space to store the result before the CALL instruction, and then remove the data from the stack afterwards.

- Activation Block: the memory portion in the stack created by the calling program.
- Activation Block Pointer: the register which stores the value of the stack pointer at the beginning of the subroutine. It is used by the invoked subroutine to access parameters and space for the result.

Example:

Calling Program	Invoked Subroutine	Stack												
... <b>PUSH</b> R1 ; <i>Result space</i> <b>PUSH</b> R19 ; <i>Second parameter</i> <b>PUSH</b> R18 ; <i>First parameter</i> <b>CALL</b> <b>subroutine</b> <b>POP</b> R0 ; <i>discard</i> <b>POP</b> R0 ; <i>discard</i> <b>POP</b> R20 ; <i>Get result</i> ...	<b>subroutine:</b> <b>IN</b> R31, 0x3E <b>IN</b> R30, 0x3D ; <i>Z ← SP</i> ... <b>LDD</b> R18, Z+3 ; <i>first parameter</i> <b>LDD</b> R19, Z+4 ; <i>second parameter</i> ... <b>STD</b> Z+5, R20 ; <i>Set result</i> <b>OUT</b> 0x3E, R31 <b>OUT</b> 0x3D, R30 ; <i>SP ← Z</i> <b>RET</b>	<table><tr><td>Return @3</td><td>(Z)</td></tr><tr><td>Return @2</td><td>(Z+1)</td></tr><tr><td>Return @1</td><td>(Z+2)</td></tr><tr><td>Value in R18</td><td>(Z+3)</td></tr><tr><td>Value in R19</td><td>(Z+4)</td></tr><tr><td>Result space</td><td>(Z+5)</td></tr></table>	Return @3	(Z)	Return @2	(Z+1)	Return @1	(Z+2)	Value in R18	(Z+3)	Value in R19	(Z+4)	Result space	(Z+5)
Return @3	(Z)													
Return @2	(Z+1)													
Return @1	(Z+2)													
Value in R18	(Z+3)													
Value in R19	(Z+4)													
Result space	(Z+5)													

Explanation of Stack:

- PUSH R1, R19, R18 to the stack: push a space for the subroutine result, then push the two parameters.
- CALL subroutine: push the return address to the top of the stack (3 bytes).
- Subroutine RET: the return addresses in the stack is removed.
- POP R0, POP R0: remove the two parameters from the top of the stack.
- POP R20: obtain the result from the stack, store it in R20.

Local Variables:

- When local variables are created inside the subroutine, and need to be allocated in the stack, they become part of the activation block. PUSH the local variables to the top of the stack.
- Reserve space in stack for local variables BEFORE coping the value of the stack pointer in Z.

### Example

High Programming Language:

- the invoked subroutine: int **count** (String str, char value)
- the instruction in calling program: result = **count** (message, 's')

Data Section	Calling Program	Invoked Subroutine
msg: <b>.string</b> "My string" result: <b>.space</b> 1, 0	<b>PUSH</b> R0 <b>LDI</b> R0, 's' <b>PUSH</b> R0 <b>LDI</b> R18, lo8(msg) <b>PUSH</b> R18 <b>LDI</b> R18, hi8(msg) <b>PUSH</b> R18 <b>CALL</b> count <b>POP</b> R0 <b>POP</b> R0 <b>POP</b> R0 <b>POP</b> R0 <b>STS</b> result, R0	<b>count:</b> <b>IN</b> R31, 0x3E <b>IN</b> R30, 0x3D <b>LDD</b> R27, Z + 3 <b>LDD</b> R26, Z + 4 <b>LDD</b> R18, Z + 5 <b>CLR</b> R20 <b>loop:</b> <b>LDD</b> R19, X+ <b>CPI</b> R19, 0 <b>BREQ</b> done <b>CP</b> R19, R18 <b>BRNE</b> loop <b>INC</b> R20 <b>JMP</b> loop <b>done:</b> <b>STD</b> Z + 6, R20 <b>OUT</b> 0x3D, R30 <b>OUT</b> 0x3E, R31 <b>RET</b>