ELEC 1601 REVIEW

Useful Website

- Reputation converter: https://tool.oschina.net/hexconvert/, https://c.runoob.com/front-end/58

Week 1

Perspectives of a computer system

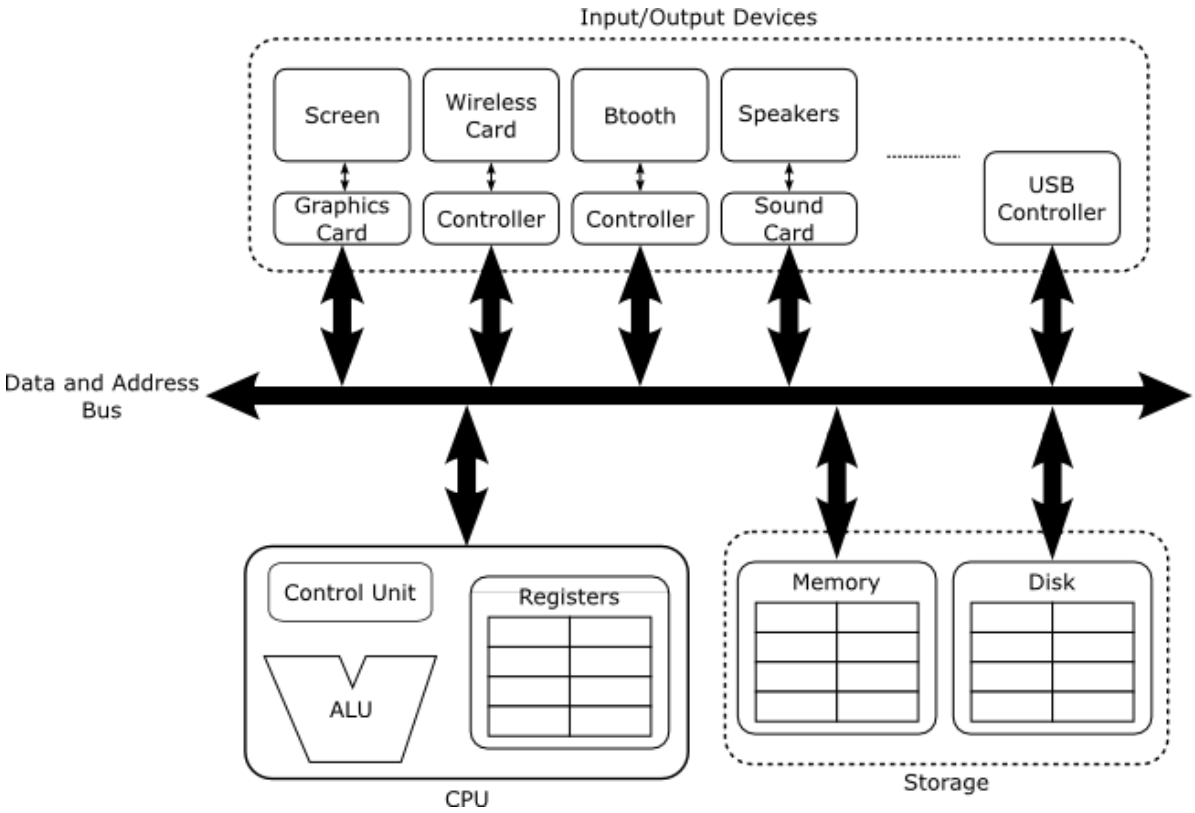
> **Compilation**: convert into a machine-code or low-level form

---

Abstraction Levels

> **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**

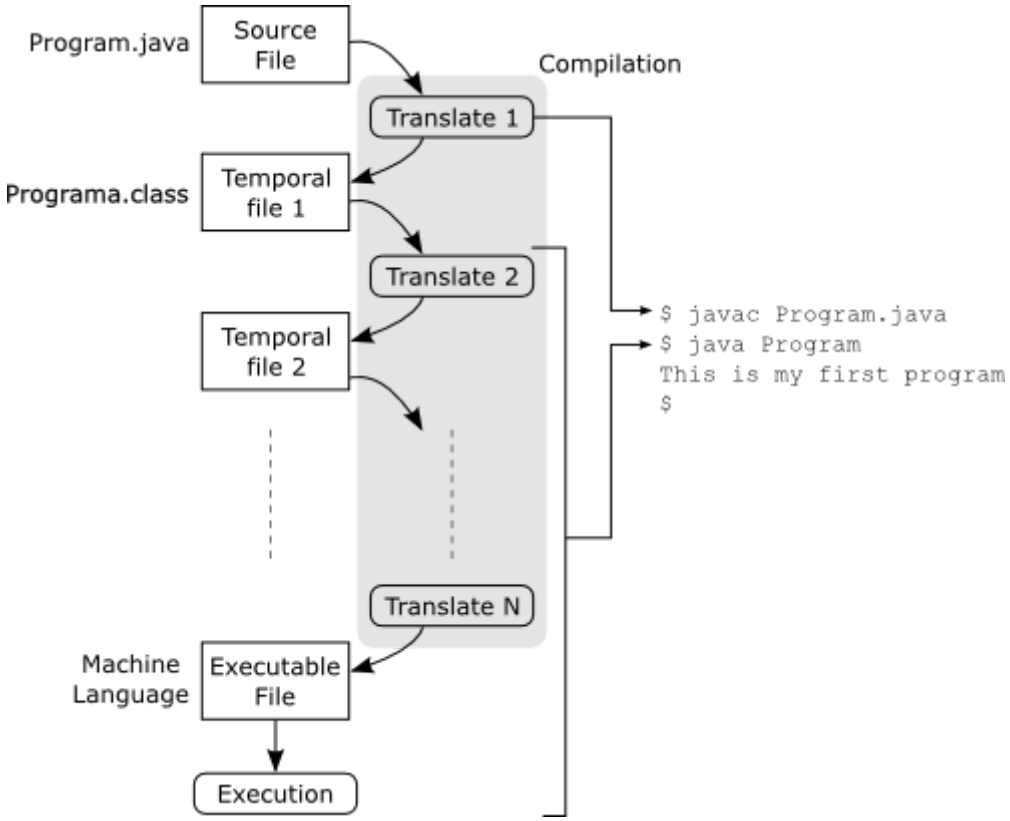
- **dynamic RAM**
- **Flash memory**
- **hard drive**
- ...

Input/output devices

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

Definition of a Program

> **Program** is a group of orders or instruction



- `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

---

Week 2

Binary Logic

> **Microprocessor**: complex digital circuit capable of executing a previously defined set of instructions encode with bits

Properties of a binary encoding

N is the **number of elements** need to be encoded, n is the **number of binary bits**

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

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

Representing number in different bases

Number represented in a **base b**:

- b digits are used to represent number, from 0 to b-1
- The number represented by b-1 is followed by number 10

$$\sum_{i=0}^{n-1}(d_i * b^i) = d_0 + (base * \sum_{i=i}^{n-1}(d_i * base^{i-1}))$$

Example:

$$218 = (108 * 2) + 1$$
$$108 = (54 * 2) + 0$$
$$54 = (27 * 2) + 0$$
$$27 = (13 * 2) + 1$$
$$13 = (6 * 2) + 1$$
$$6 = (3 * 2) + 0$$
$$3 = (1 * 2) + 1$$
$$1 = (0 * 2) + 1$$
$$\therefore 217 = 0b11011001$$

- **Binary**: 0b
- **Octal**: base 8, 0o, three bits a group
- **Hexadecimal**: base 16, 0x, four bits a group

Size of an encoding

Use n bits to encode natural numbers

$$Range = [0, 2^n - 1]$$

If number larger than $2^n - 1$, overflow appears

Encoding integers

Sign and magnitude

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

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

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

2s complement

> **2s complement** only use 1 bits to represent 0, all zero, most significant bits to represent sign, 0 as positive and 1 as negative

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

Decimal to Binary

If the number is **positive**

- Simply translate it to base 2 encoding

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**

- 1. Replace 1 with 0, replace 0 with 1
  2. Add 1
  3. Translate the resulting number to base 10, take its negative value
- Use $ABS(N) - 2^n$

Bit extension

- If the number is **positive**, adding zeros to the most significant bits
- If the number is **negative**, adding ones to the most significant bits

---

Encoding Real Number with Floating Point Representation

- When a digital circuit manipulates real numbers it introduces **an error**, in some cases error could be 0

Floating point

- The part to the **right** of the point is called mantissa (or significand)
- How the exponent and mantissa are encoded can vary between encoding systems

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

1. turn base 10 to base 2
2. move radix point until the most significant bit of mantissa is 1
3. record sign, mantissa and exponent

Range

14 bits are allocated as 1 for the sign, remaining 7 for the mantissa and 6 bits for the exponent (2s complement)

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

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

Underflow appears when trying to represent a very small number

---

Maximum error

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

---

Encoding set of symbols

The set ual-1

Instruction = Operation + Operand

Operation	8 bit number	8 bit number

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

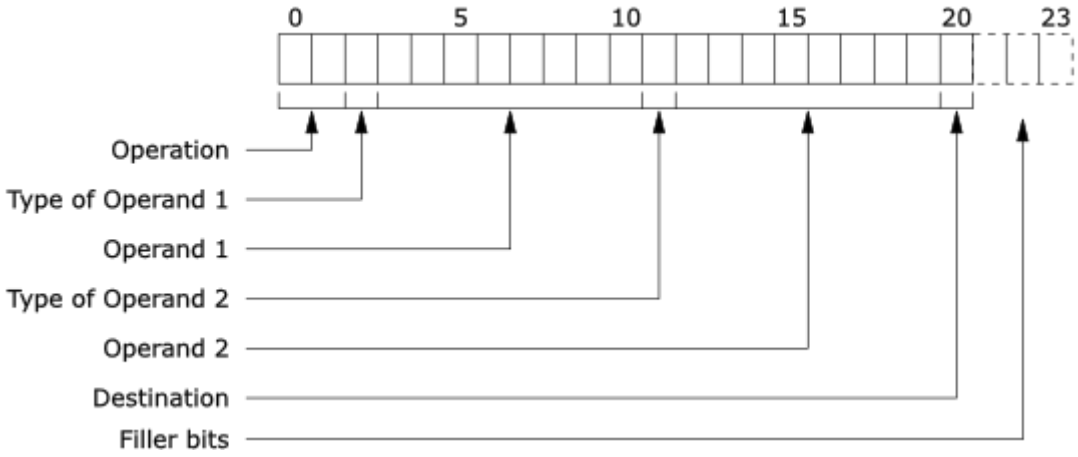
Operation	8 bit number	8 bit number	Location

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
- If type of operand is 1: ignore follow 7 bits, read the **last bit** to obtain the value location

---

Instruction with variable size

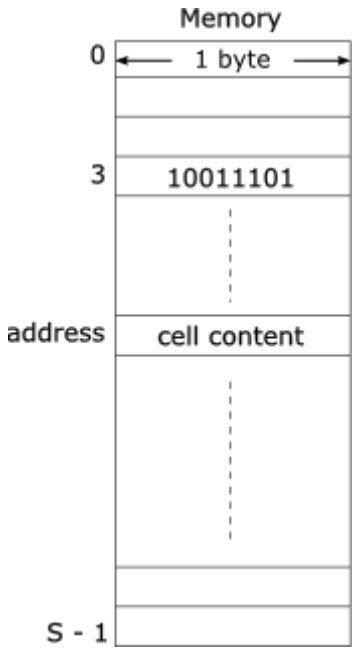
Fixed format: all instructions have the same length

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

Week 3

RAM Memory



- RAM(**random access memory**)
- Most common 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

SRAM

Static RAM: Data is stored until it is overwritten or the power is stopped, **transistor**

DRAM

Dynamic RAM: **Capacitors**, is charged it is storing a one, need to recharge over certain time interval, if a read operation is done over a cell, it is refreshed

SRAM vs DRAM

- DRAM is simpler, so higher capacity and cheaper
- DRAM have to be refreshed, slow speed of read/write operation
- DRAM Refresh all cells continuously before lose charge
- SRAM is faster

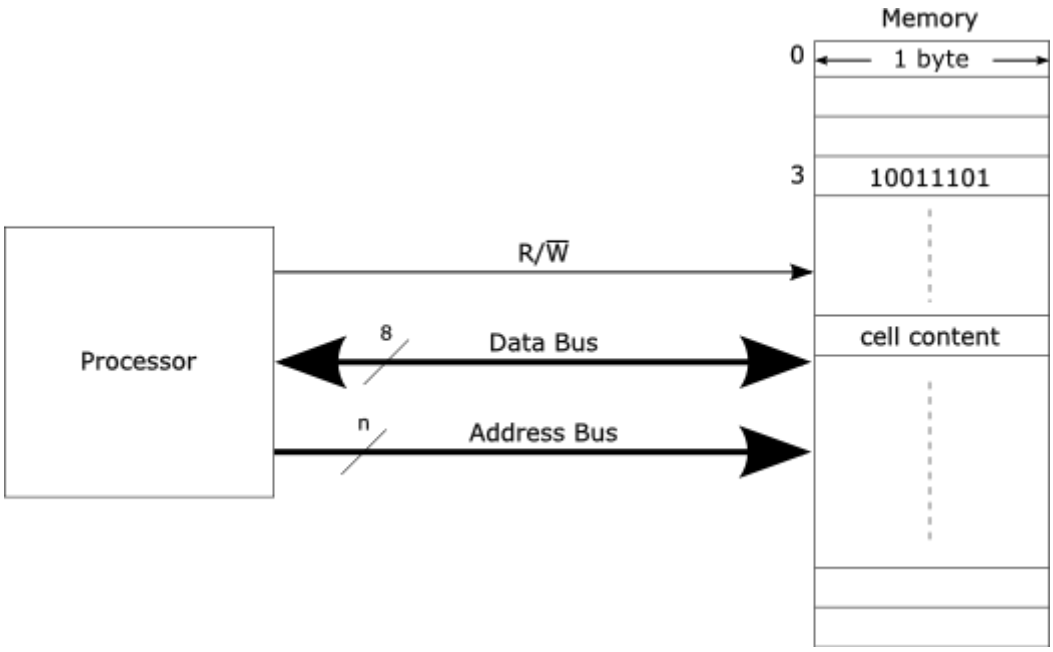
Memory operations

> **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 **undefined** 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

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

Connection between memory and processor



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

- **Address bus**: Send addresses of operation to memory chips
- **Data bus**: Send data
  - Writing: processor -> memory chip
  - Reading: memory chip -> processor
- **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

Data Storage

Size of Java basic data types

Type	Value	Size(bit)
Boolean	true, false	1
byte	Integer	8
char	Unicode	16
short	Integer	16
int	Integer	32
long	Integer	64
float	IEEE-754	32
double	IEEE-754	64

Boolean

Two methods:

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

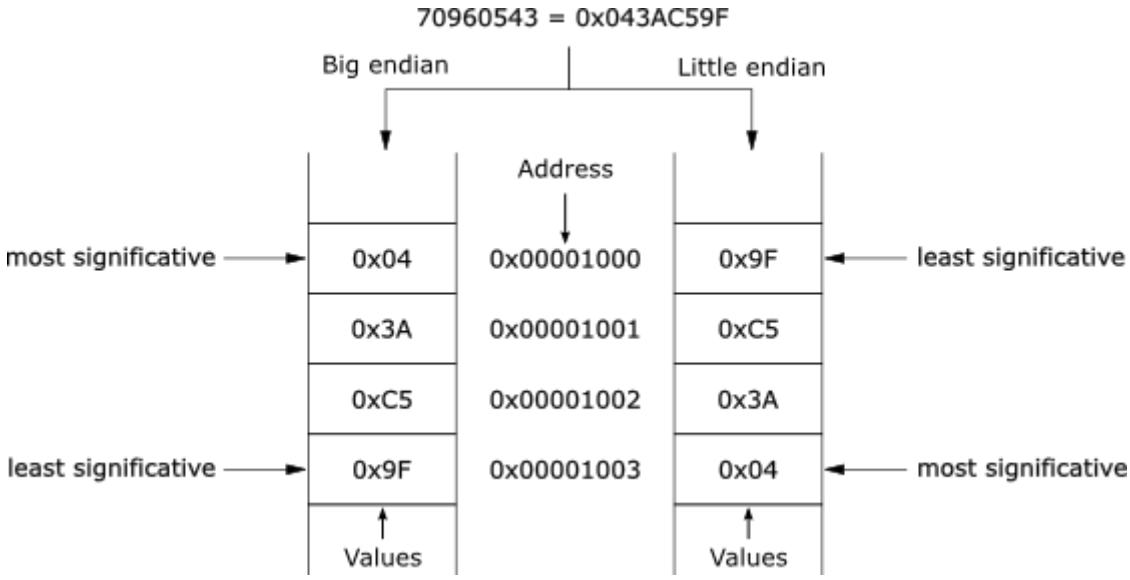
A sequence of characters will occupy **consecutive cells** in memory

ASCII, 1 bytes

UTF-16, two bytes for 1 char

Integers and natural numbers

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.

Little\Big endian policy only applies when the data is stored in memory

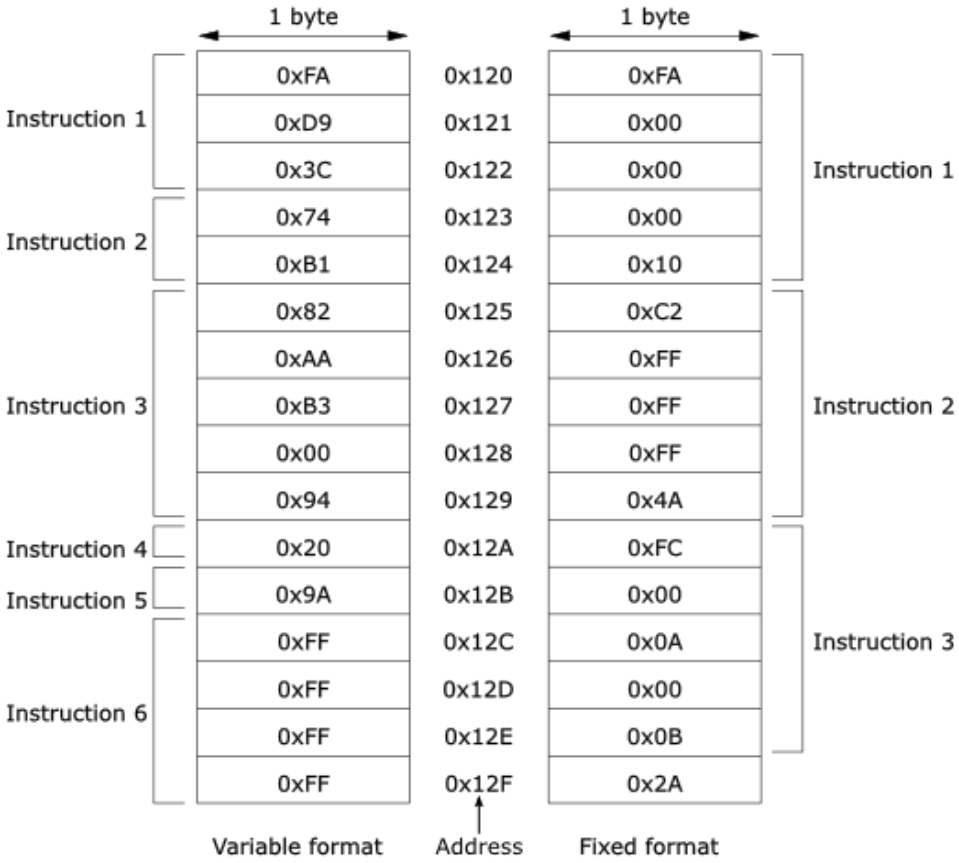
---

Storage for machine instructions

Two types:

- **Fixed length**: 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

    1. First obtain one instruction from memory
    2. Interprets its content, deduces its size
    3. Add ahead to address and obtain the next one



No need to distinct little and big endian

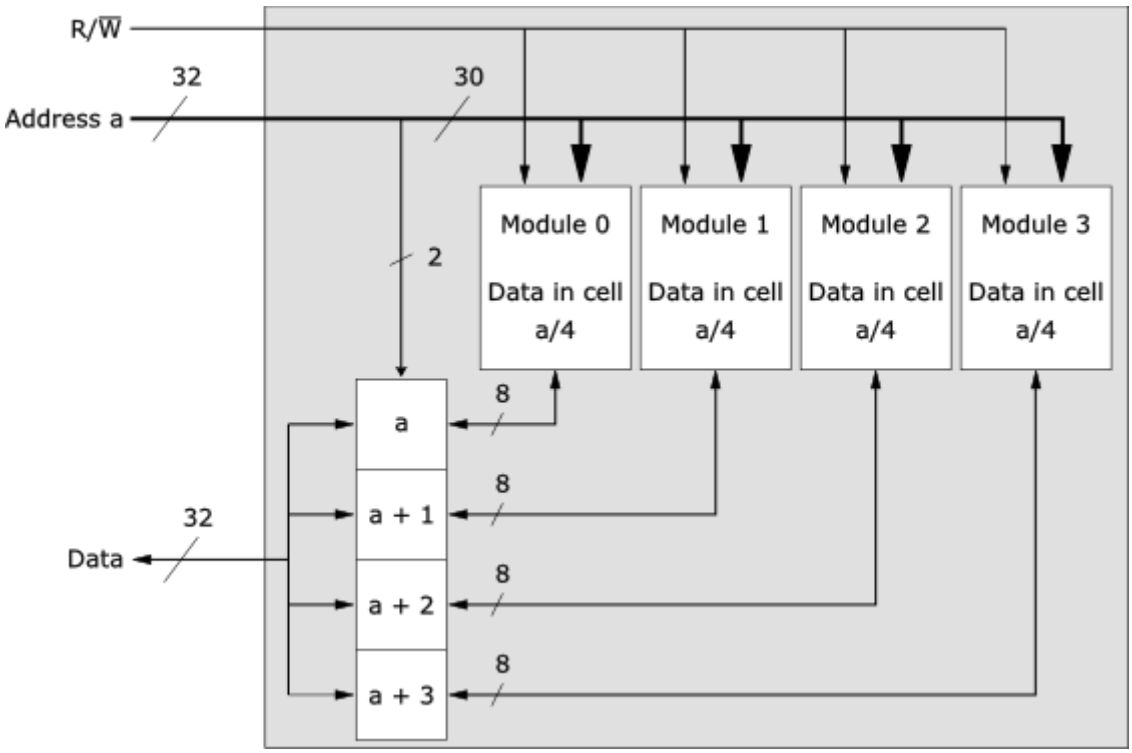
Size of the read and write operations in memory

Bottleneck: read or write operation is much **slower then processor**

Solution: Read and write **several cells** at the same time

Assume 4 cell a modules

- 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(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
- 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

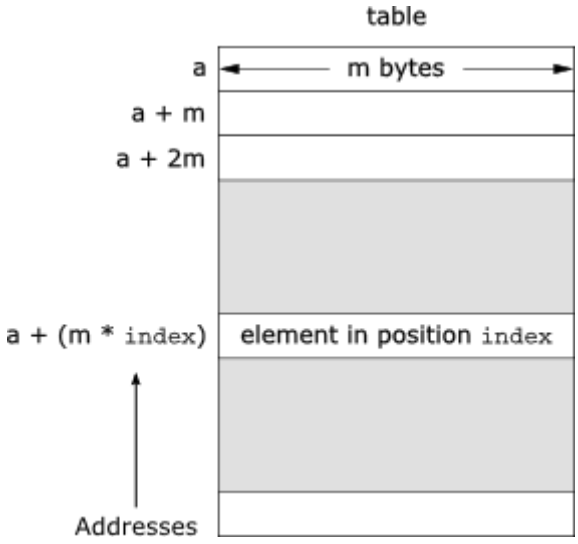


multiple byte operations: If a memory is capable of reading and writing data in groups of x bytes, then instruction `Write(address, value)` will write value in the address and x position after that, same rule in `Read`

Storing an Array

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

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



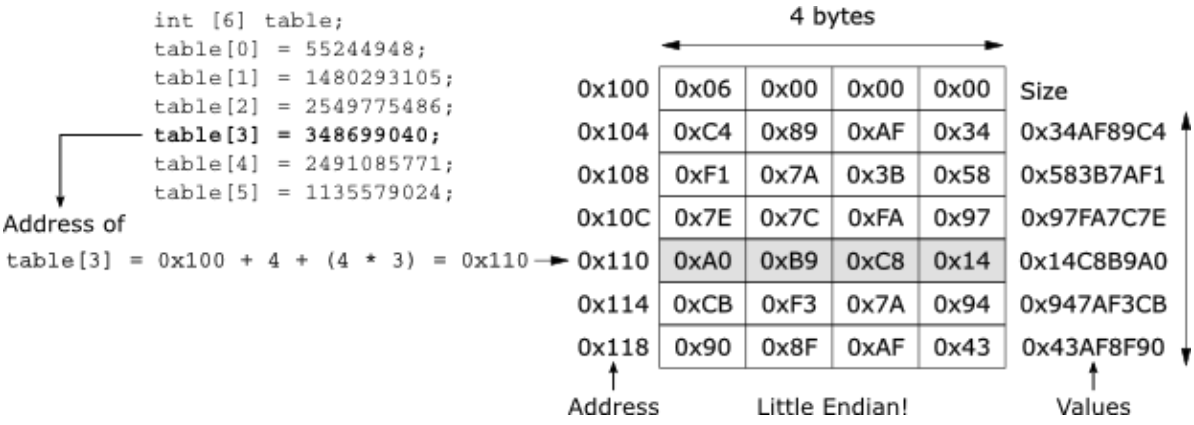
Two way to locate the end of the array:

- 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

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**

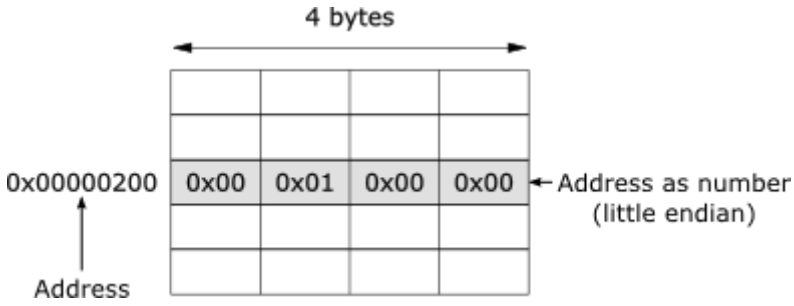


Step:

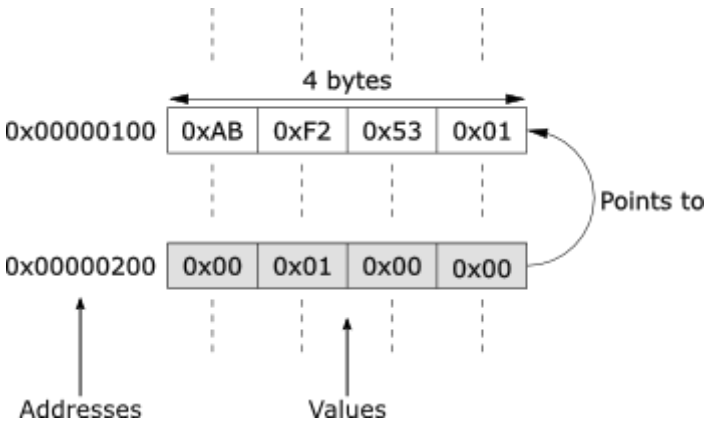
1. Obtain the integer s stored in position d (size)
2. Check that $0 \leq i$
3. Check that $i < s$
4. Calculate the address of the element as $d + 4 + (t * i)$

Storing memory addresses

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



Use the value which stored in the memory as an address to read the specific value, this kind of access is called **an indirection**

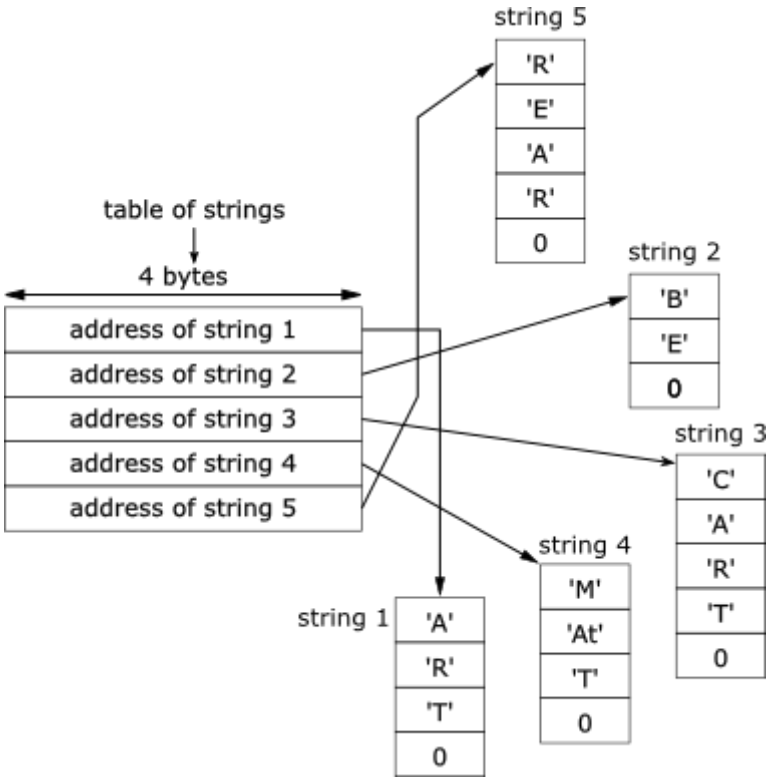


Though this chain structure we can establish double indirection, triple indirection and etc

Examples of indirection

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



Week 4

Boolean Expressions

Operation	Symbols	Result
Conjunction	$and, *$	true when both operands are one
Disjunction	$or, +$	true when any operand is one
Negation	$not, x', \bar{x}, \neg x$	The opposite of the operand

Boolean function: given the value for a set of symbols, return 1 or 0

Identical Boolean Expressions

Two expressions are equivalent if they have the same truth table

---

Simplification of Boolean Expressions

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	xy = yx, 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:

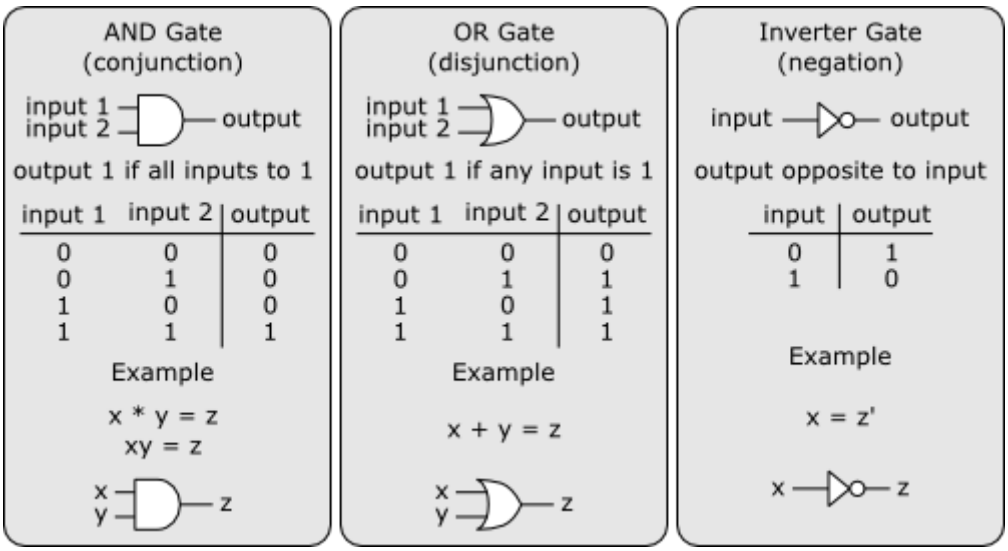
1. select the row which result is equal to 1
2. write the product of symbols and negate those with value 0
3. add all

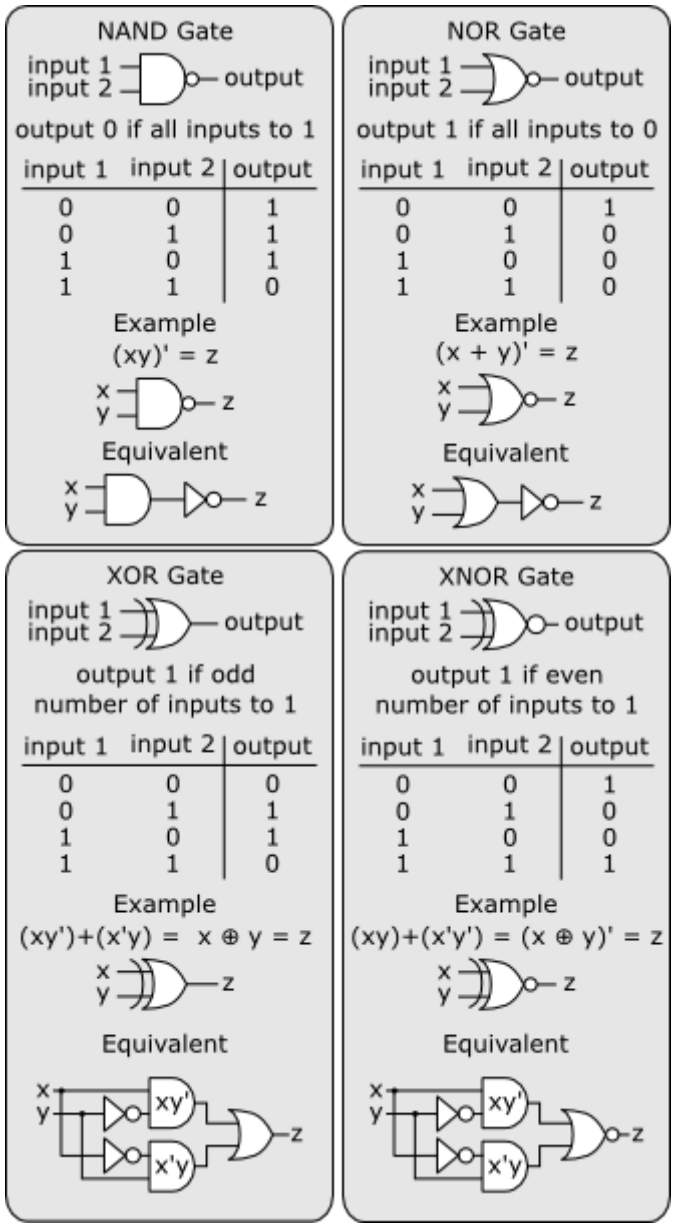
Products of Sum:

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

Logic Gates



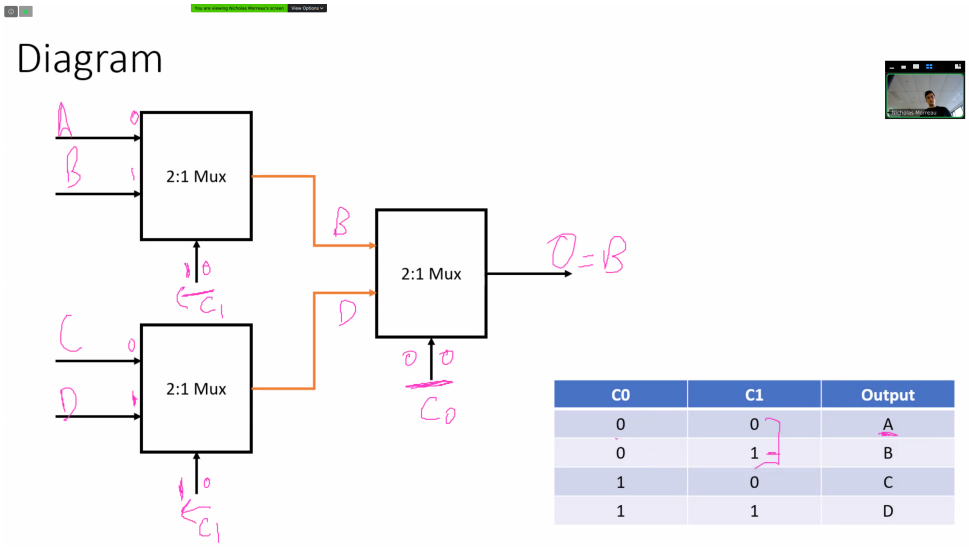


Mention: XOR and XNOR output depend on the number of inputs, even or odd

Combinational circuits: Output of one gate becomes the input of another gate, cannot be a cycle

Multiplexer

When **the control signal is one**, the output takes the value of the input a, otherwise, it takes the value of input b.

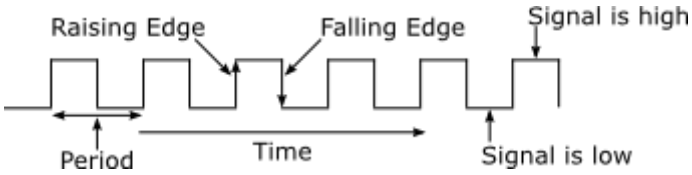


C0	C1	Output
0	0	A
0	1	B
1	0	C
1	1	D

Week 5

The 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**: signals that design when to pay attention to the value of the inputs, when to maintain the value of output

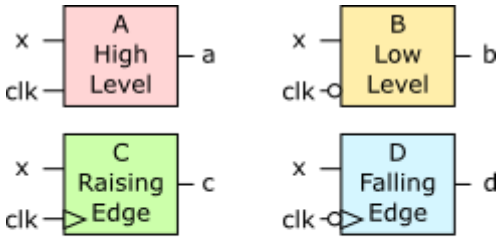


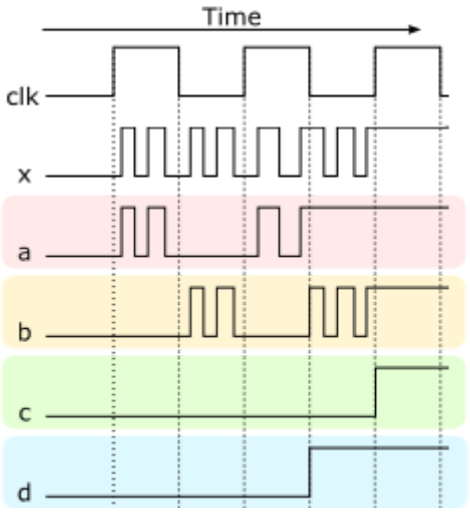
- Over time, the value of this signal oscillates from high to low
- The transitions occur almost instantaneously, **falling edge**, **rising edge**
- Period of clock, represented by T, from low to high, and back to low, the frequency of a clock represented by f, $f = 1/T$, 5 GHz clock means th signal 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**

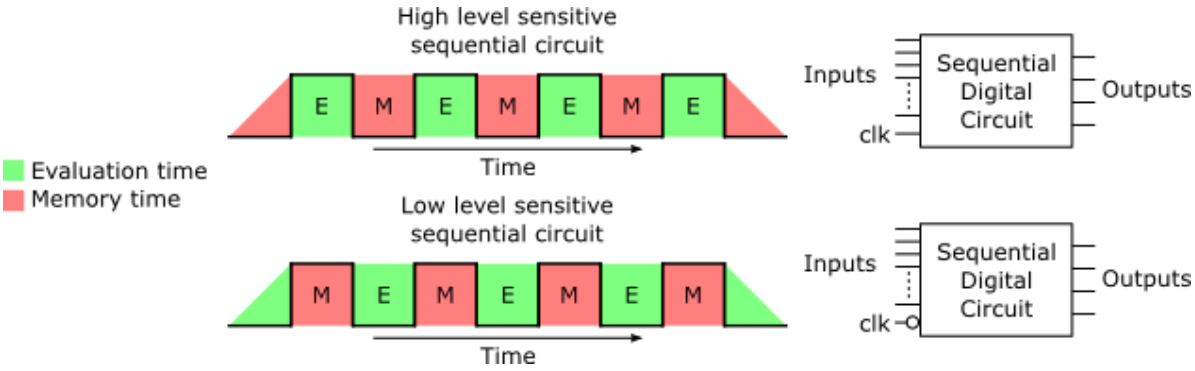
Sensitivity in Sequential Circuits





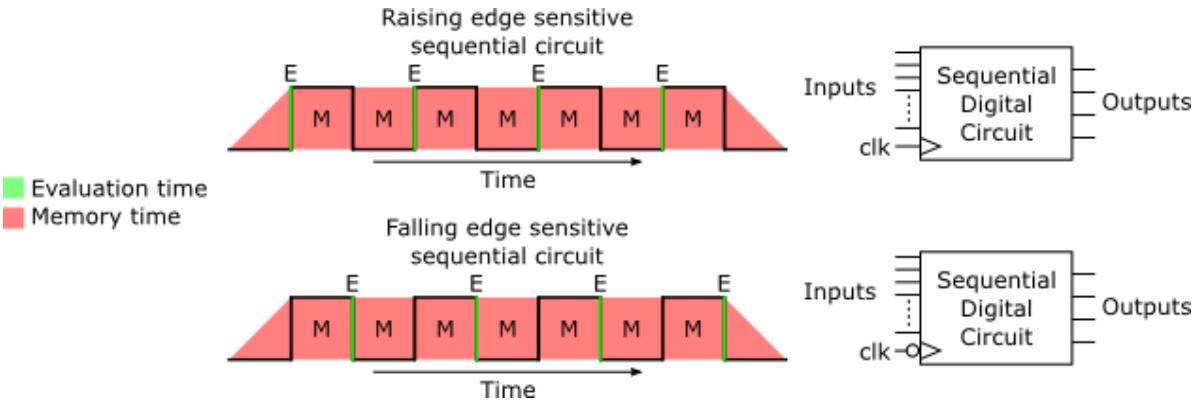
Level-Sensitive Sequential Circuits

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



Edge-Sensitive Sequential Circuits

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



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

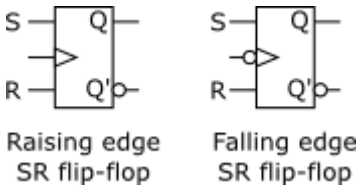
SR Filp-Flop

- Has two inputs，S (set)and R (reset)

Input		**Output**
S	R	Q_{t-1}
0	0	$Q_t(no\ change)$
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
- When set and reset are both set to 1, no way to tell the ouput

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



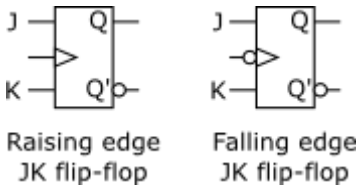
JK Filp-Flop

- 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

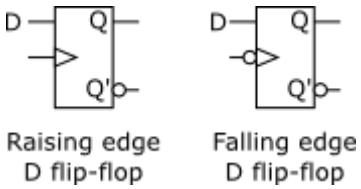


D Filp-Flop

- Has one input, D

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

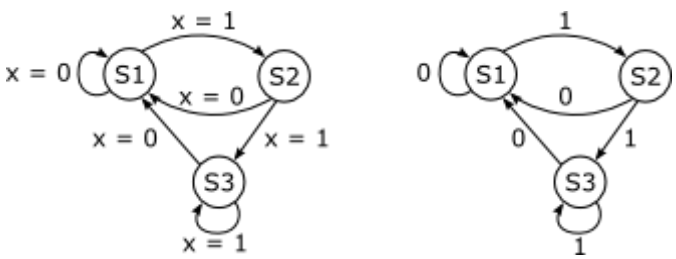
D	Q_{t+1}
0	0
1	1



Finite State Machines

- In a given state and after the clock edge, it transitions to a new state
- Edge can have no expression on it

Designing FSMs

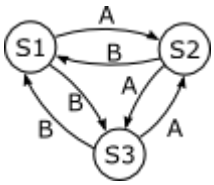


1. Identify the different **states**
2. Draw each of the states with a name in a circle

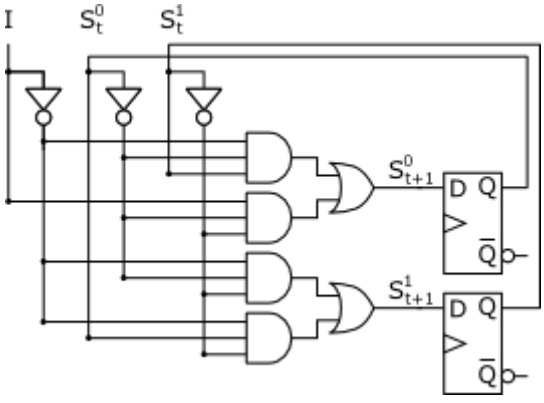
3. Identify the system **inputs**
4. Consider how will the **state change** for every possible combination of input
5. Label each transition connecting two states with the input values

From a FSM to 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**, in below case,
   $S_t^0 S_t^1 = 00 = state1, S_t^0 S_t^1 = 01 = state2, S_t^0 S_t^1 = 10 = state2$
4. Truth table

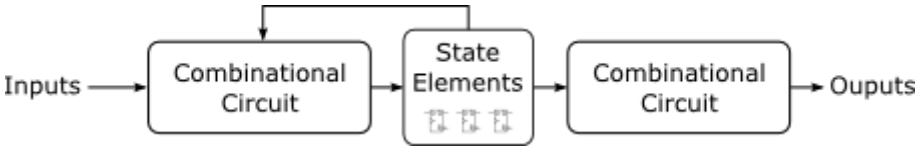


Inputs			Outputs	
I	S_t^0	S_t^1	S_{t+1}^0	S_{t+1}^1
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	X	X
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	X	X

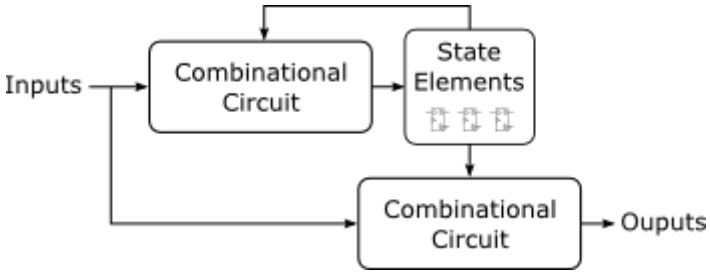


Moore and Mealy

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



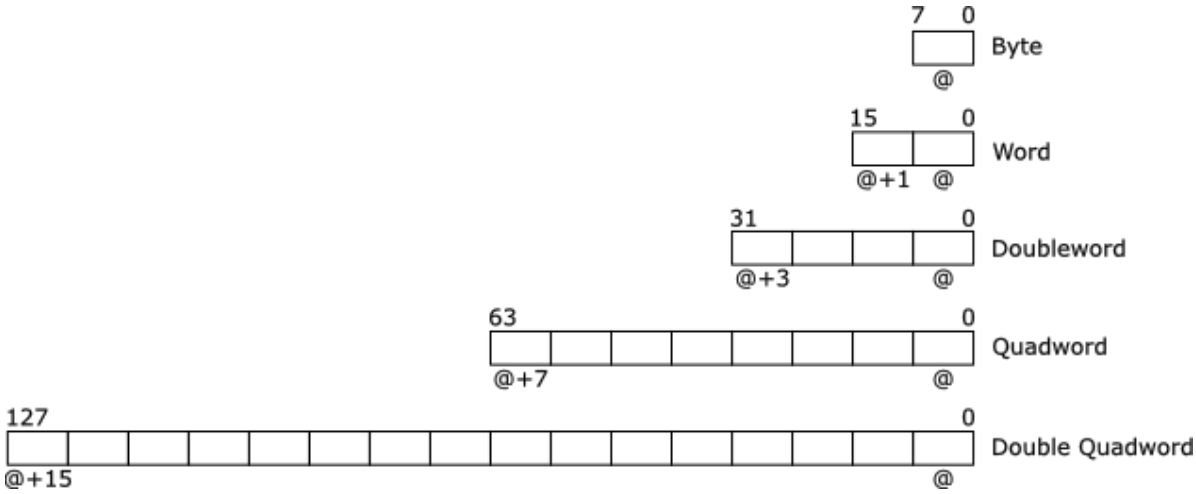
Mealy: Depend on both the current state and the value of the inputs



Week 6

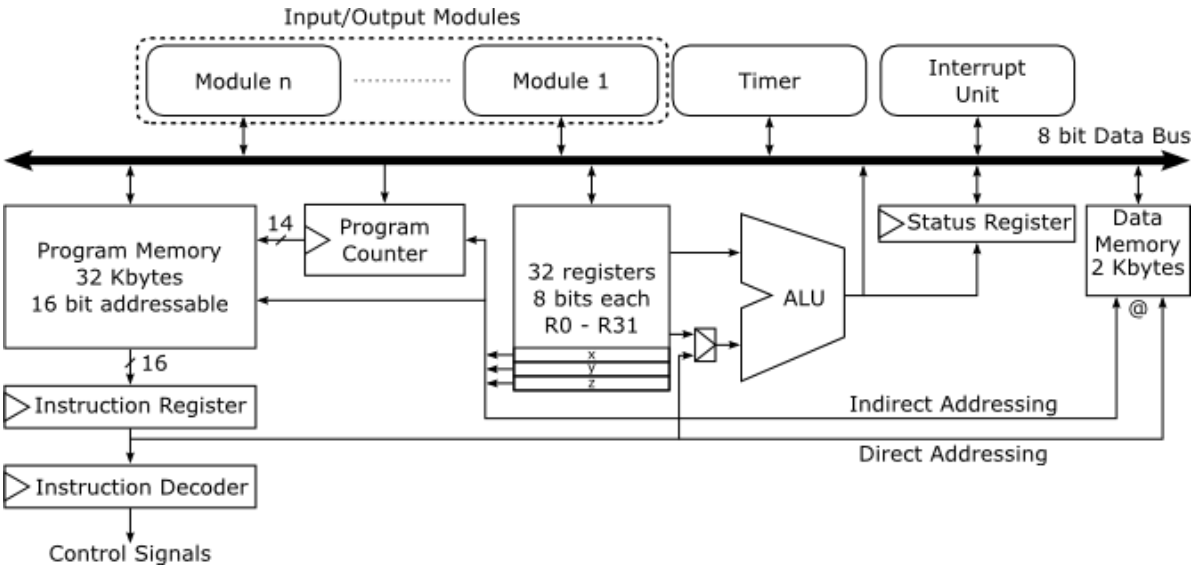
The Execution Environment

Name	Bytes
Byte	1
Word	2
Doubleword	4
Quadword	8
Double Quadword	16



Architecture: A set of micro-controllers that are capable of executing the same machine instructions

The data path



The blocks are all 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.

Program and Data Memories

Assume that the program memory is 32 Kbytes($2^{15} bytes$), the data memory is 2 Kbytes($2^{11} bytes$)

AVR architecture contains two types of memory

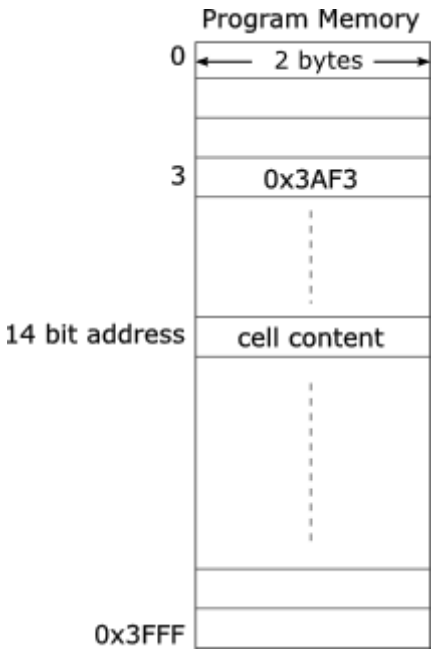
1. **Program Memory**
2. **Data Memory**

Machine instructions are obtained from the **Program Memory** and stored in the **Instruction Register**

Program Memory

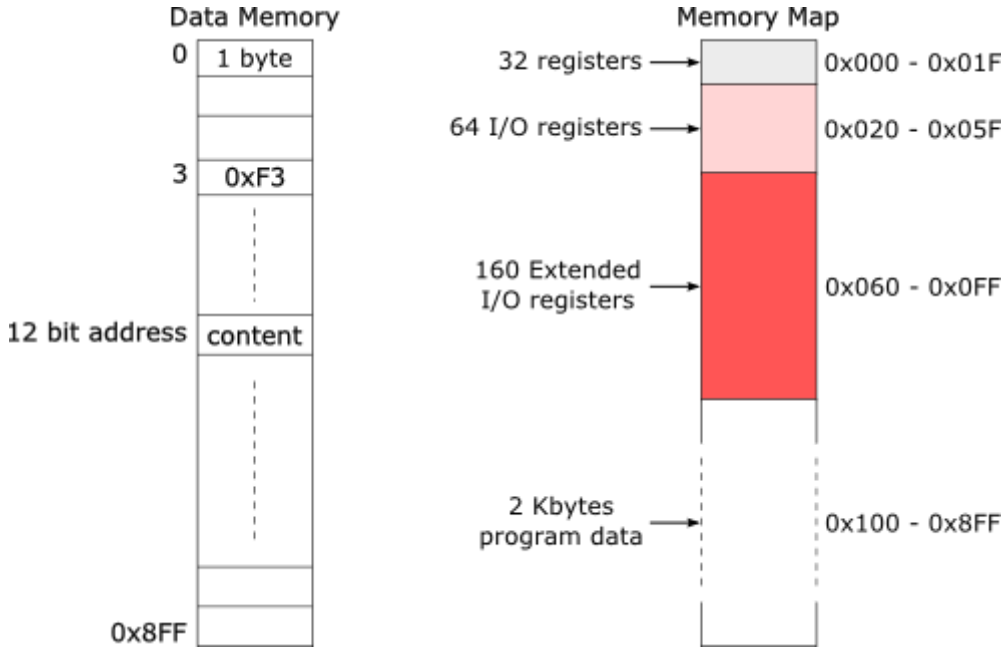
The program memory is a **flash memory**, **2 byte addressable**, 14 bits address

Store the code to execute



Data Memory

The data memory is a **static RAM**, **1 byte addressable**. (The first 256 position is designed to access **32 general purpose registers, 64 additional I/O register, 160 extended I/O registers**, from 0x000 to 0x0FF)



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

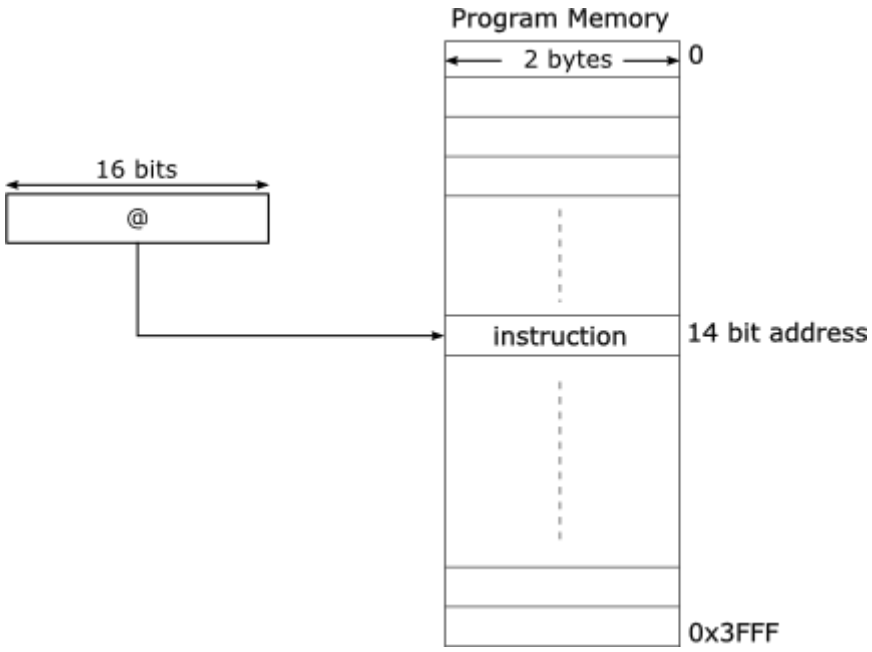
Third memory

EEPROM (Electronically Erasable Programmable Read Only Memory)

Register

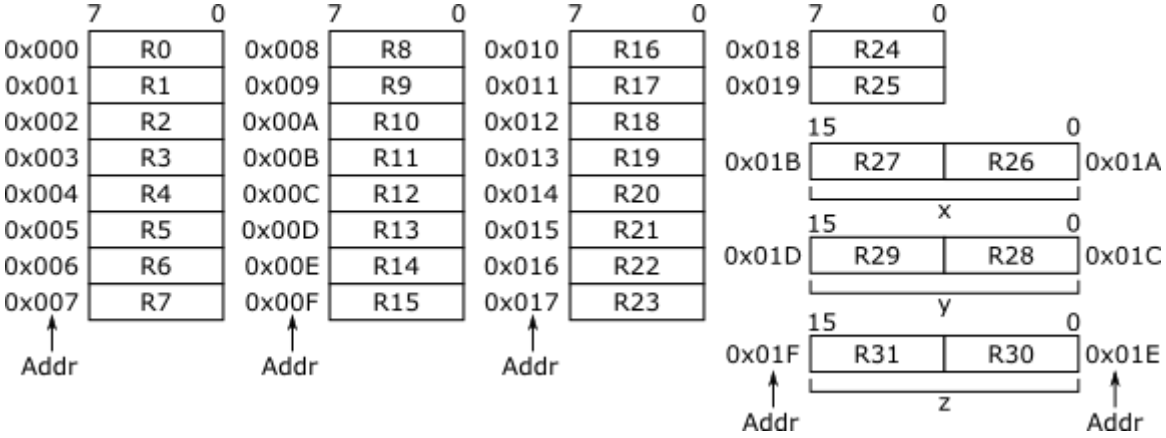
Program counter

- Point to program memory
- Contain the address in the **program memory** of the next instruction to be executed
- Some special instructions modify this register



General Purpose Register

- The AVR architecture has 32 8-bit general purpose rigister, known as **register file**
- Named 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

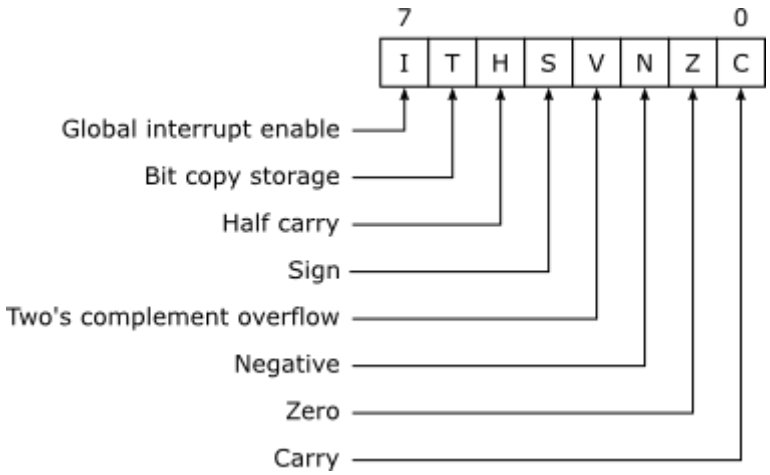


Status Register

- The number of bits and conditions stored in this type of register is **dependent on the architecture**
- Reflect special situations when they occur, other instructions make decisions based on these values
- Refresh **every time** an operation is executed

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, **has produced an overflow**
5. **Sign flag (S)**:indicates the sign of the result of the latest arithmetic or logic operation
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



Arithmetic/Logic Unit(ALU)

Combinational circuit capable of performing operations of three types

1. arithmetic
2. logical
3. Bit level functions

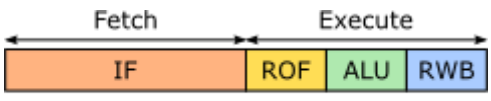
Operands

1. From register file
2. One from register file, second from the **instruction register** (a field inside the encoding of a machine instruction)

Capable of multiplying two signed or unsigned integers.

The Execution Cycle

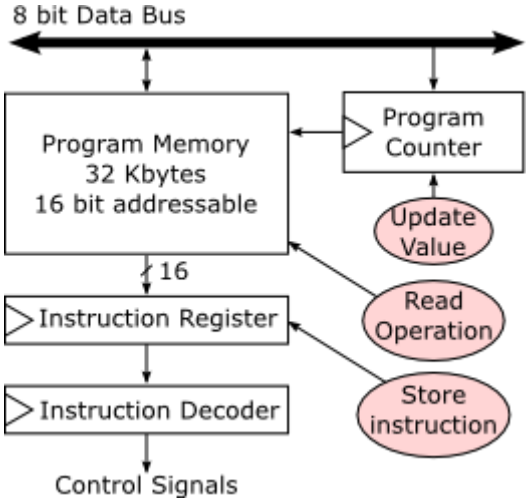
- Instruction Fetch: Obtain the instruction from the **program memory** and stored in the **instruction register**
- Execution:
    - Register Operand Fetch
    - ALU Execution
    - Register Write Back



Instruction Fetch: IF
Register Operand Fetch: ROF
ALU Execution: ALU
Result Write Back: RWB

Instruction Fetch

- Obtains the next instruction from **program memory**, read the address in address memory which stored in the **program counter** , stored it 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

---

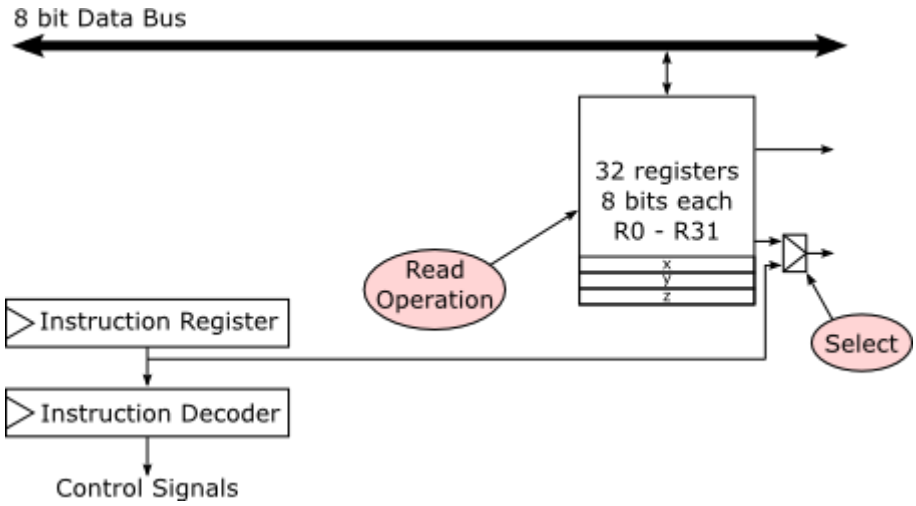
Execution

Register Operand Fetch

Valid for instructions that 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.

  (**parameters output from register, result 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 part of the instruction
- Some instructions 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.



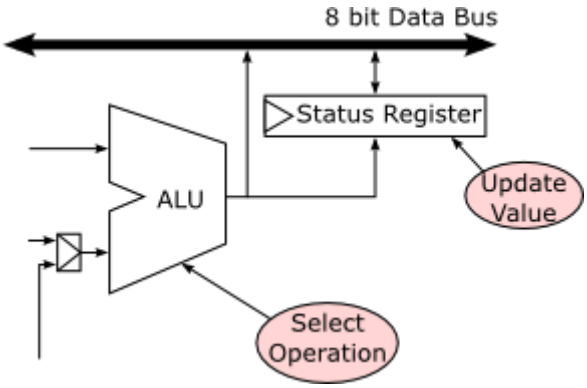
Select part is to select the operand is either from registers or from instruction register directly

---

ALU Operation Execute

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

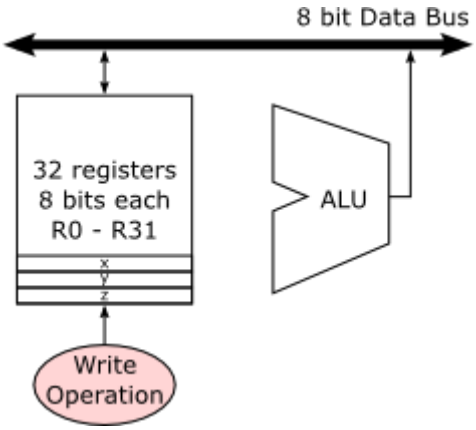
- The **control signals** select the appropriate operation and the **result** is produce at the output



Result write back

Only those result produced by the ALU needs to be stored in the register file

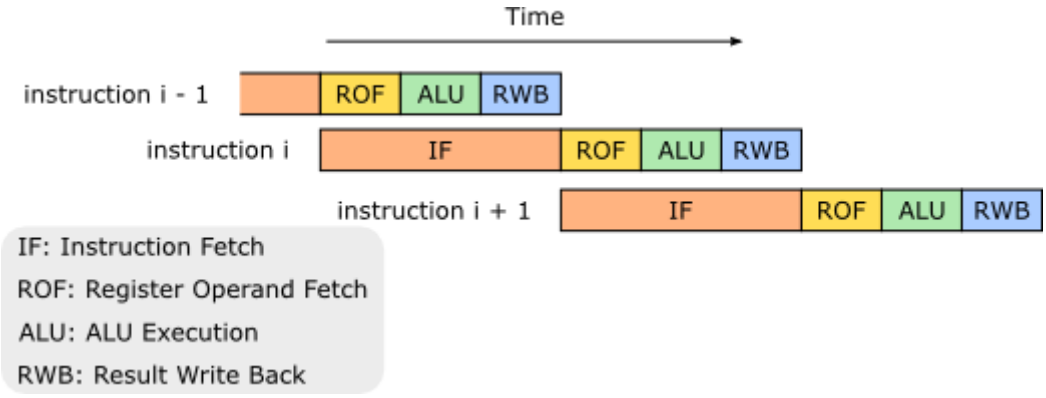
- ALU write result on to the data bus.
- register selects the destination for the result



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

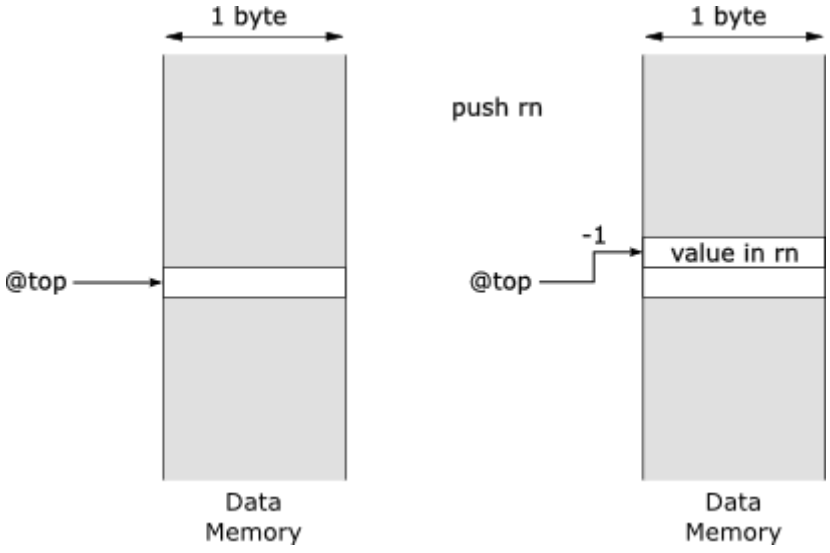
In AVR, the instructions are restricted to use registers, and the **data size is always 1 byte**

Stack Instructions

`push Rn`

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

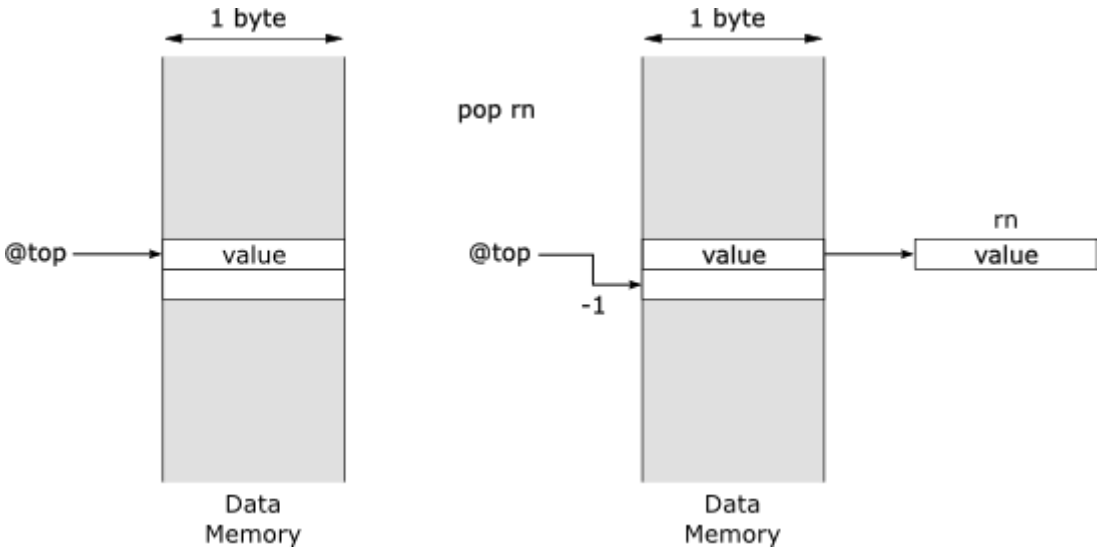
1. @top = @top - 1
2. Given register is written in @top



`pop Rn`

Store the content at the top of the stack into the register

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



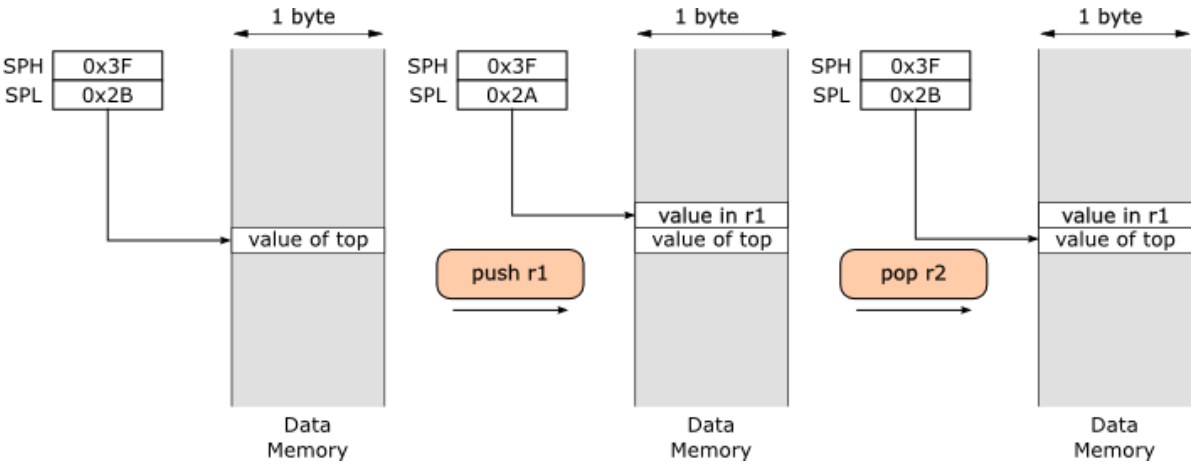
- Aside from the stack instructions, the instructions to call and return from a subroutine also use the stack

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.

The stack pointer

- Implicit operand: push and pop doesn't specify the destination.
- In AVR, the stack pointer 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 the **last position in data memory**)

Week 7

Types of Instruction Sets

Instruction set architecture or ISA: The set of instructions a microprocessor can execute

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

Complex Instruction Set Computers (CISC)

- Larger set of instructions
- Use **several** operands and require **multiple** memory accesses
- Execute a smaller number of instructions, larger time to execute

Reduced instruction set computers (RISC)

- Fewer set of instructions, very **simple operation**
- Faster execution and decode, simpler structure
- Execute a larger number of instructions

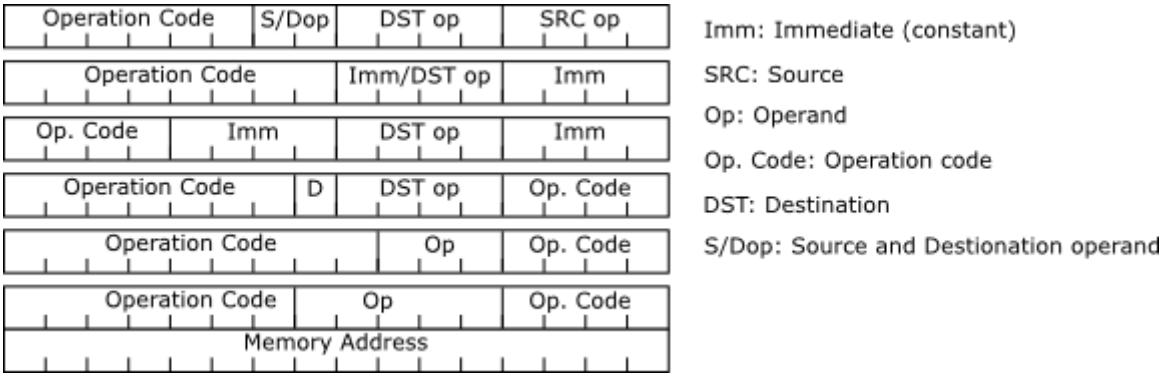
Fixed length format and Variable length format

- Fixed length format: Every instruction has the same size (**RISC**)
- Variable length format: All instruction to be encoded with different number of bits (**CISC**)

Instruction Format of AVR Architecture

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`



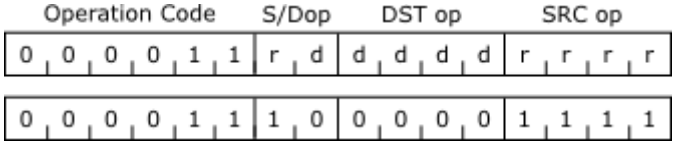
Operation code: Every instruction must have some bits to encode the type of operations

---

Operands are one of the 32 general purpose registers

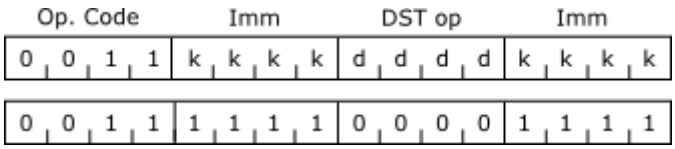
Require five bits per operand. These bits are **not necessarily in contiguous positions** in the contiguous position in the instructions

```
ADD Rd, Rr
ADD R0, R31
```



Number as an operand instead of a register

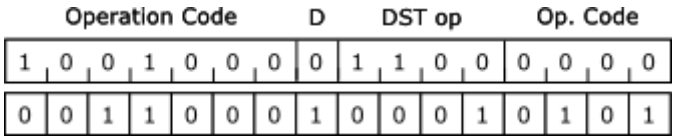
```
CPI Rd, K
CPI R16, 255
```



This instruction is used to compare **immediate operands and register**, this instruction can only use registers that index **from 16 to 31**, because it has only 4 bits to encode the immediate operands, 8 bits for value K ($0 \leq K \leq 255$)

Memory addresses as an operand

```
LDS Rd, k
LDS R12, 12565
```



In AVR, a memory address is 16 bits long, so it need another memory to store the operands, the decoding stage detects that is `LDS` instruction so double access to memory

---

Assembly Language

- **Same** instructions, same operands and formates as the machine language
- alphanumeric representation, executed as part of machine language

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

Subset of Instructions of the AVR Architecture

Instructions to transfer data

- `MOV Rd Rr`, makes a copy of one register into another
- `LDI Rd, K`, loads an **8 bits** constant value in one of the **general purpose registers R16 to R31**
- `LD Rd, X/Y/Z`, loads the content from memory address which contained in one of the **16 bits registers**
  - The value of memory address can be post-incremented or pre-decremented
  - `X+` means $X \leftarrow X + 1$ **after** execution, `-Y` means $Y \leftarrow Y - 1$ **before** execution
- `LDS Rd, k`, leads one byte from the data space

- `LDD Rd, Y/Z + q`, loads one byte from the memory address, the memory is **adding the register and the displacement**, $0 \leq q \leq 63$
- `ST X/Y/Z, Rr`, stores the **byte in register** into the address pointed by `X/Y/Z`,
  - The value of memory address can be post-incremented or pre-decremented
- `STD Y/Z + q, Rr`, stores the **byte in register** into the address pointed by `Y/Z + q`, $0 \leq q \leq 63$
- `STS k, Rr`, stores the byte in register in an address directly, the address is a label **previously defined**
- `PUSH Rr`
- `POP Rd`

Arithmetic Instructions

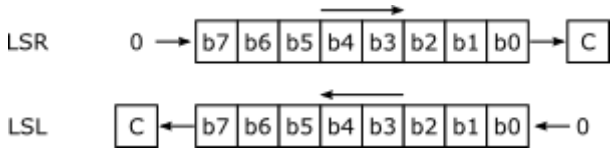
- `ADD Rd, Rr`
- `SUB Rd, Rr`
- `SUBI Rd, k`, subtracts an 8 bits constant from the register
- `INC Rd`, adds one to the content
- `DEC Rd`, subtracts one to the content
- `NEG Rd`, changes the sign of the value in a register
- `MUL Rd, Rr`

Logic Instructions

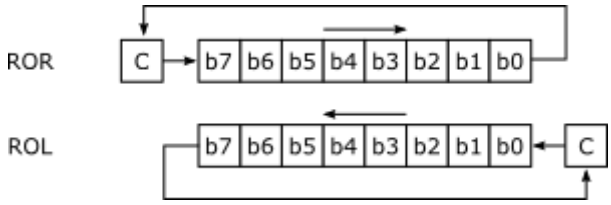
- `AND Rd, Rr`
- `OR Rd, Rr`
- `EOR Rd, Rr`, performs the logical exclusive OR

Shift and Rotate Instructions

- `LSR Rd`, `LSL Rd`, same as multiplying **signed and unsigned** values by 2



- `ROR Rd`, `ROL Rd`, instruction shift the carry bit C into bit 7/bit 0, can be used to multiply multi-byte numbers



- `ASR Rd`, shifts all bits in Rd one place to the right, Bit 7 is held constant, bit 0 is loaded into the C flag

Compare Instructions

- `CP Rd, Rr`, performs a comparison **between two registers**, result modifies the flags of the status register
- `CPI Rd, K`, performs a comparison **between a register and a 8 bits constant**

Jump and Branch Instructions

- `JMP k`, jumps to an address `k` in the **program memory**, $0 \leq k < 2^{22}$
- `BREQ k`, `BRNE k`, jump to an address k, if the previous comparison or operation resulted in a zero or one result (**check the Z flag**, Z==1 is equal, Z==0 is not equal), $-64 \leq k \leq 63$
- `BRSH k`, `BRLO k`, condition is unsigned(**If it is negative, change to positive, -64 == 65**), check the C flag and if C==0, it is same or higher, otherwise it is smaller, $-64 \leq k \leq 63$, if
- `BRGE k`, `BRLT k`, condition is signed, check the C flag and if C==0, it is greater or equal, otherwise it is lower, $-64 \leq k \leq 63$

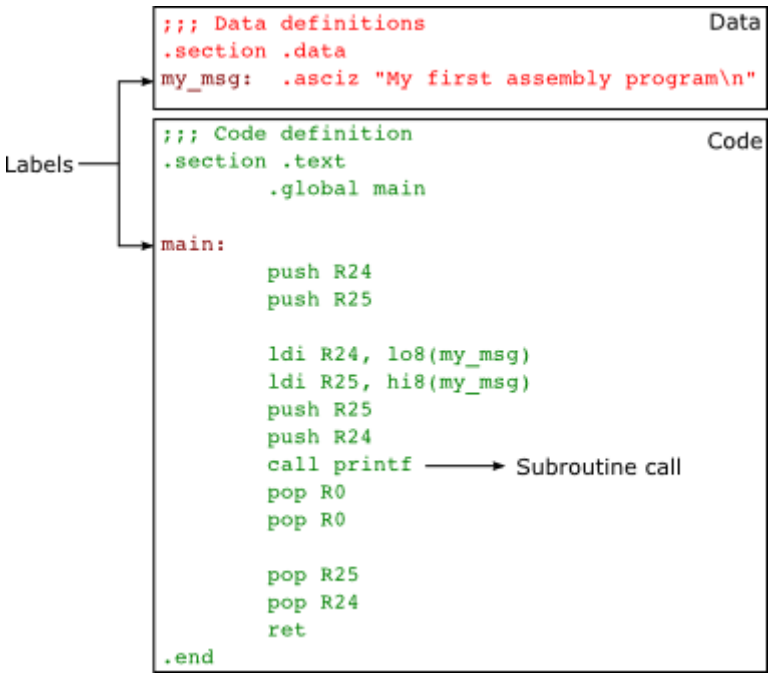
Input and Output Instructions

- `IN Rd, A`, reads the value of an input port and store that in the register
- `OUT A, Rd`, writes or stores the data in the register to the output position

---

Week 8

Creating an Executable Program from Assembly Code



- `.section .data` doesn't translate into any instruction, just a mark, called **directives**
- `my_seg` is a label
- `.section .text` tells the assembler that the data definitions have finished
- `main` is global symbol

Basic structure

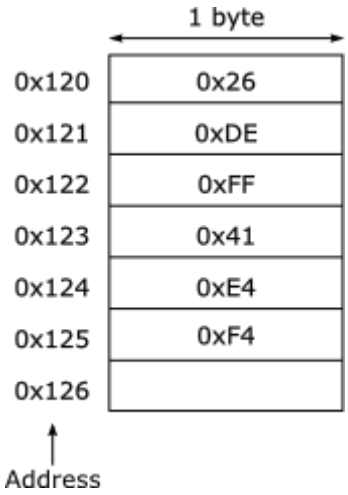
```
;;; Data definitions go here
.section .data

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

main:
        ret
.end
```

Data Definition

All data definitions in an assembly program must be included after `.section .data`

Definition of byte

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

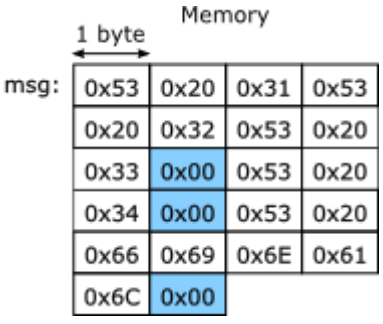


38 is stored in label data, others are stored in **consecutive location** after that

Definition of String

- `.ascii` each character is encoded in one byte and stored in consecutive memory position
- `.asciz` same, but each string is encoded with an extra byte **at the end** with value `0x00`
- `.string` same as above

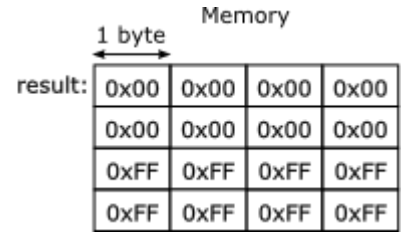
```
msg:    .ascii "S 1", "S 2"
        .asciz "S 3", "S 4"
        .string "S final"
```



Definition of Space

`.space` reserve memory space with the **second value**

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

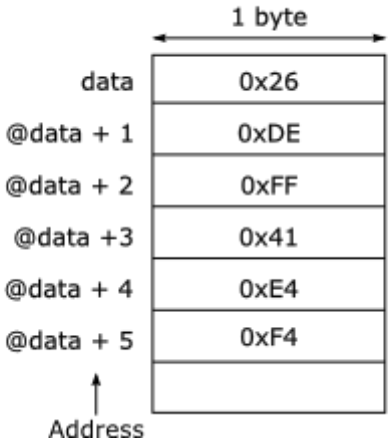


---

label

Can be used for both **data and code addresses**, represents the **address** of the first of these value

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



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

`lo8()` return 8 least significant bits of the **address** of the label

To access memory at a certain distance from the label, need to load label in to X/Y/Z registers then use `LDD` or `STD`

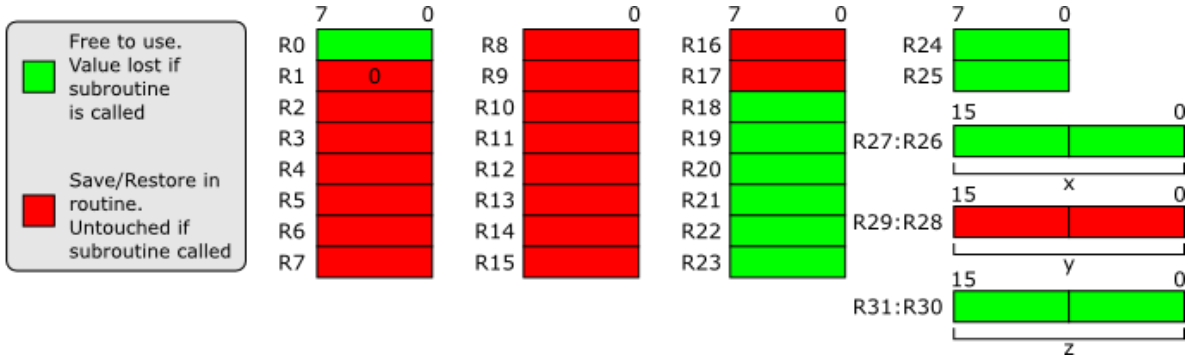
Restriction

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**
- R18 to R27, R31:R30 **may not** be saved and restored
- If a function returns an interger, this must be placed in R25:R24 at the end of the subroutine



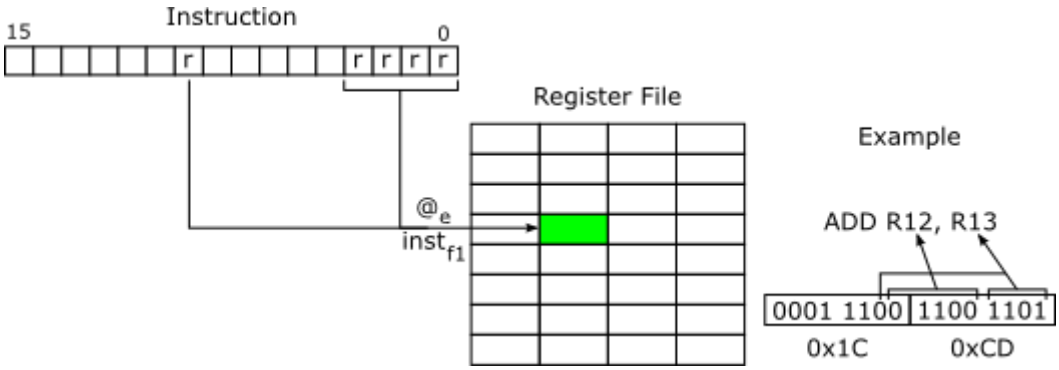
Addressing modes

- 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.

Register Direct

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

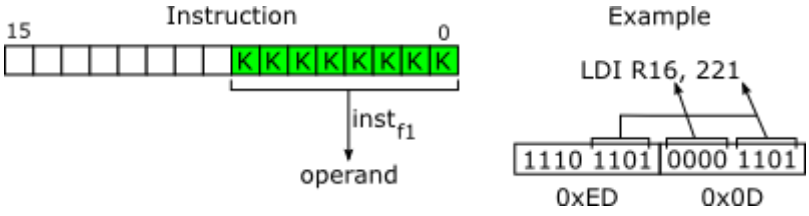
$$@_e = @inst_{f1}$$
$$operand = R_{f1}$$



Immediate

Specify a constant as part of the instruction, **provides the operand directly**, only allow to use registers R16 to R31

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

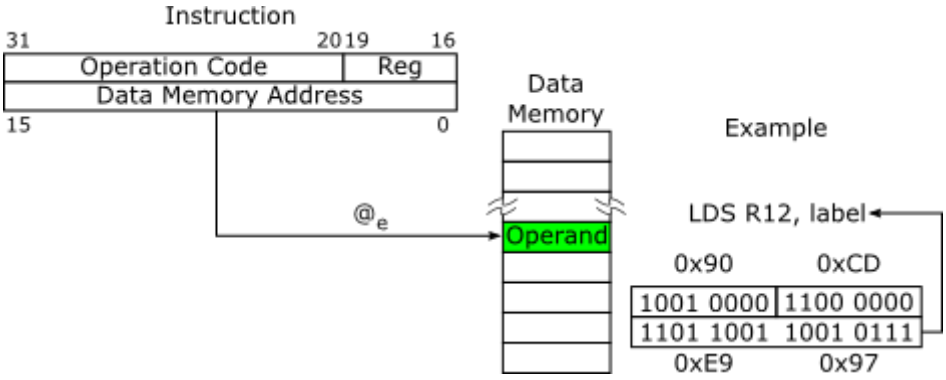


Data direct

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, $[@inst + 1]$ because the address is stored in the next position



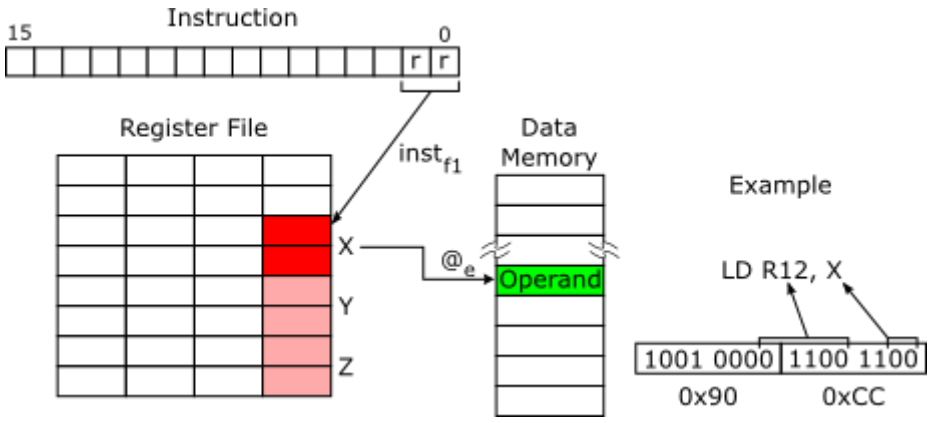
Require three memory access, the program counter need to be **updated twice**

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

Two least significant bits to cover the three options

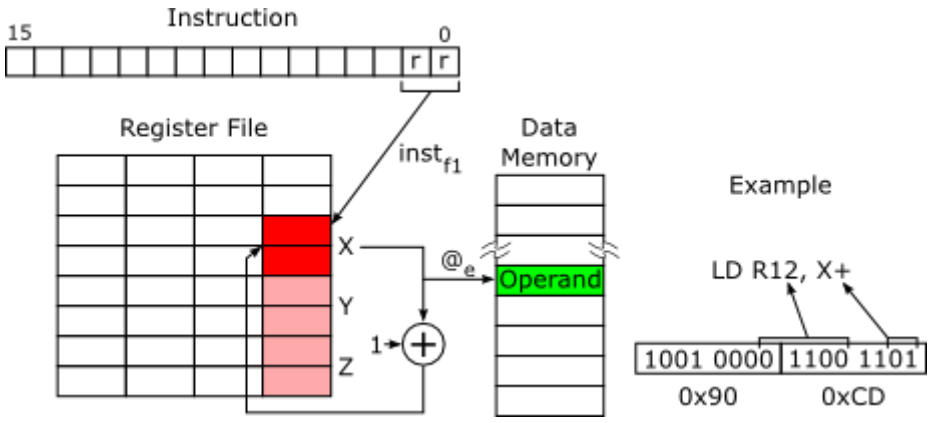
$$@e = R_{f1}$$
$$operand = DMEM[R_{f1}]$$



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

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

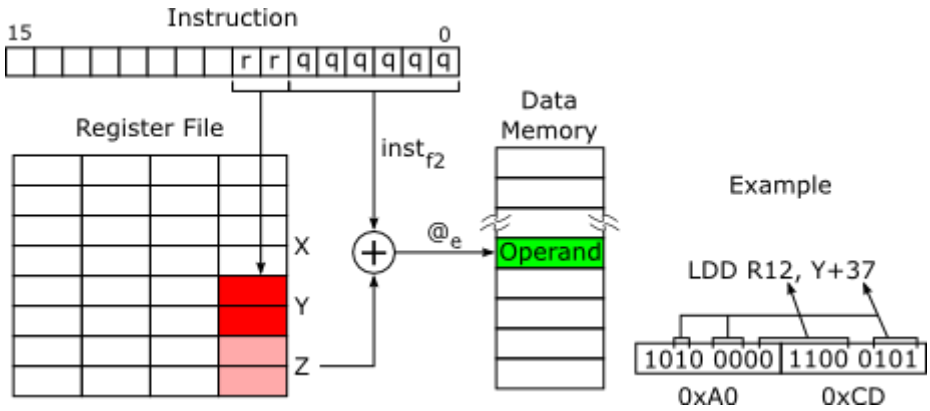
`push`

---

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 $0 \leftarrow d \leftarrow 63$

Week 10

Translation of High Level Programming

if-then-else

```
;;; Data definitions go here
.section .data
;;; Code definition goes here
.section .text
    .global asm_function

asm_function:
        LDI R25, 2
        LDI R24, 1
        CP R24, R25     ; if 1 > 2
        BRSH firstOne   ;   firstOne();
        LDI R24,-1      ; else
        LDI R25,-1      ;    return -1;
        ret

firstOne:
        LDI R24,1       ; return 1;
        CLR R25
        ret

.end
```

---

switch

```
;;; Data definitions go here
.section .data
my_log_0:  .asciz "this is case 0\n"
my_log_1:  .asciz "this is case 1\n"
my_log_2:  .asciz "this is case 2\n"
my_log_default:  .asciz "this is case default\n"
;;; Code definition goes here
.section .text
    .global asm_function

asm_function:
        LDI r25, 2
        LDI r24, 1
        ADD r24, r25          ;switch(x+y)
        CPI r24, 0            ;case 0:
        BREQ case_0           ;  case_0() break
        CPI r24, 1            ;case 1:
        BREQ case_0           ;  case_1() break
        CPI r24, 2            ;case 2:
        BREQ case_0           ;  case_2() break
        JMP case_default      ;default:
        ret

case_0:
        ldi r26, lo8(my_log_0)
        ldi r27, hi8(my_log_0)

        push r27
        push r26
```

```
        call printf
        pop r0
        pop r0

        ret

case_1:
        ldi r26, lo8(my_log_1)
        ldi r27, hi8(my_log_1)

        push r27
        push r26
        call printf
        pop r0
        pop r0

        ret

case_2:
        ldi r26, lo8(my_log_2)
        ldi r27, hi8(my_log_2)

        push r27
        push r26
        call printf
        pop r0
        pop r0

        ret

case_default:
        ldi r26, lo8(my_log_default)
        ldi r27, hi8(my_log_default)

        push r27
        push r26
        call printf
        pop r0
        pop r0

        ret

.end
```

---

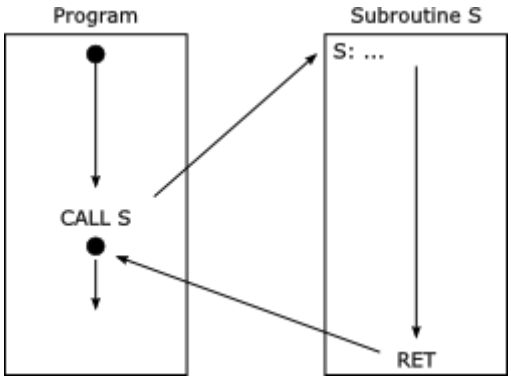
while/for

```
;;; Data definitions go here
.section .data
my_log:  .asciz "In the loop\n"
i:       .byte 0
;;; Code definition goes here
.section .text
    .global asm_function

asm_function:
loop_start:
```

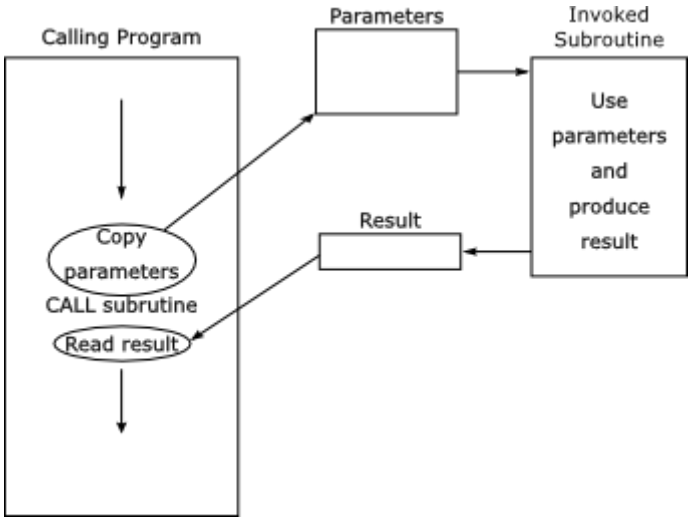
```
        LDS R24, i              ;while(i<10)
        CPI R24, 10
        BRSH done

        LDI R18, lo8(my_log)    ;  print("In the loop\n")
        LDI R19, hi8(my_log)
        push R19
        push R18

        JMP ppp

done:
        ret
ppp:
        CALL printf
        pop R0
        pop R0

        LDS R24, i               ;  i++
        INC R24
        STS i, R24
        JMP loop_start

.end
```

Subroutine Execution

- It avoids redundant code
- It promotes the division of tasks
- It promotes code encapsulation



The address used by the `RET` instruction depends on the location of the previous `CALL` instruction, obtains the **top of the stack**



Passing Parameters and Returning Result through Registers

- Parameters: Place from `R25` decreasing to `R8`
- Return: always in `R25:R24`

```
;;; Data definitions go here
.section .data
;;; Code definition goes here
.section .text
    .global asm_function

asm_function:
        LDI R25, 1
        LDI R24, 2
        CALL subroutine
        INC R24
        CLR R25

subroutine:
        MOV R5, R25
        MOV R4, R24
        ADD R4, R5
        MOV R24, R4
        CLR R25
        RET

.end
```

- Limited number of registers, but efficiency

Passing Parameters and Returning Result through Memory

```
;;; Data definitions go here
.section .data
p1:     .space 1, 0
p2:     .space 1, 0
result:     .space 2, 0
;;; Code definition goes here
.section .text
```

```
    .global asm_function

asm_function:
        LDI R25, 1
        LDI R24, 2
        STS p1, R25
        STS p2, R24
        CALL subroutine
        LDI R28, lo8(result)
        LDI R29, hi8(result)
        LD R24, Y
        INC R24
        LDD R25, Y+1
        RET

subroutine:
        LDS R19, p1
        LDS R18, p2
        ADD R18, R19
        MOV R24, R18
        CLR R25
        LDI R28, lo8(result)
        LDI R29, hi8(result)
        ST Y, R24
        STD Y+1, R25
        RET

.end
```

- The size of parameters is unknown, can't use in recursive functions

---

Passing Parameters and Returning Result through the Stack

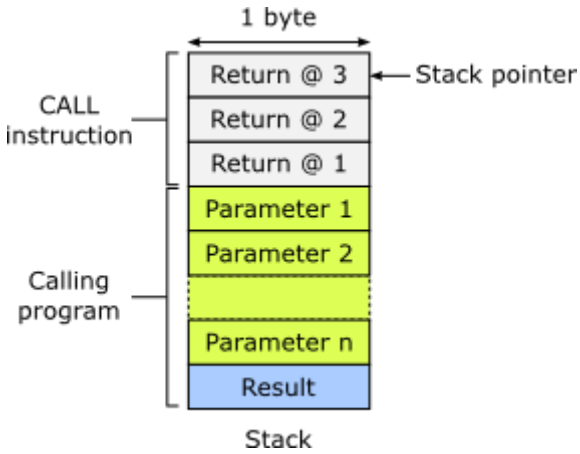
```
;;; Data definitions go here
.section .data
;;; Code definition goes here
.section .text
    .global asm_function

asm_function:
        LDI R25, 1
        LDI R24, 2
        PUSH R1                 ; Result space
        PUSH R25
        PUSH R24
        CALL subroutine
        POP R0
        POP R0
        POP R24
        INC R24
        CLR R25
        RET

subroutine:
        IN R31, 0x3E            ; address of the pointer
        IN R30, 0x3D
```

```
        LDD R24, Z+3           ; return address occupies 3 bytes in the stack
        LDD R25, Z+4
        ADD R24, R25
        CLR R25
        STD Z+5, R24
        OUT 0x3E, R31
        OUT 0x3D, R30
        RET

.end
```

- The memory portion created in the stack with this method is called the **activation block**
- The pointer is called **activation block pointer**
- The return address occupies 3 bytes in the stack
- Reserve position for result before starting a subroutine

---

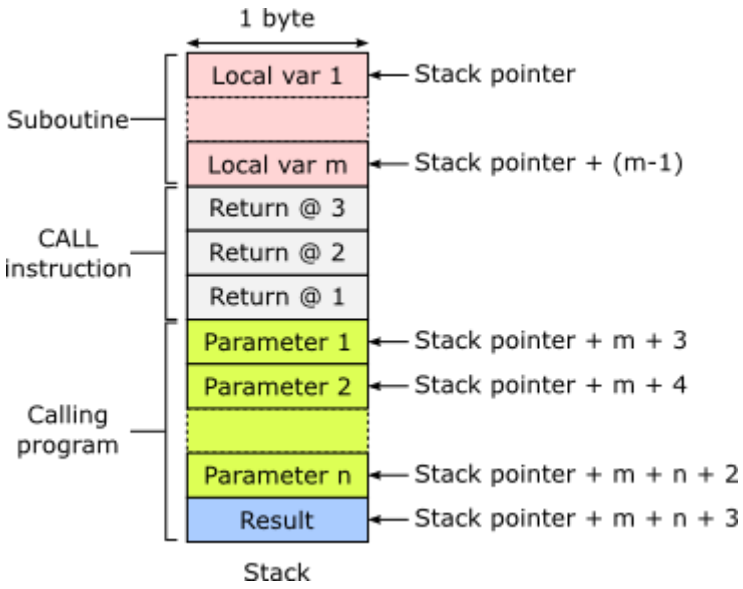
activation block



- Call function automatically push the return address to the stack, the size is 2 or 3 bytes depending on the processor model

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

- The local variables are created while executing and disappear afterwards



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`