Freie Universitat Berlin
Erasmus Program

10 ECTS

Computer Architecture

Professor:
Larissa Groth

Autor:
Filippo Ghirardini

Winter Semester 2024-2025

Contents

Computer Architecture

Autor: Ghirardini Filippo

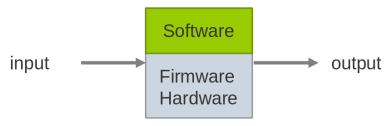
Winter Semester 2024-2025

1 Introduction

1.1 Computer

A computer is made of three components:

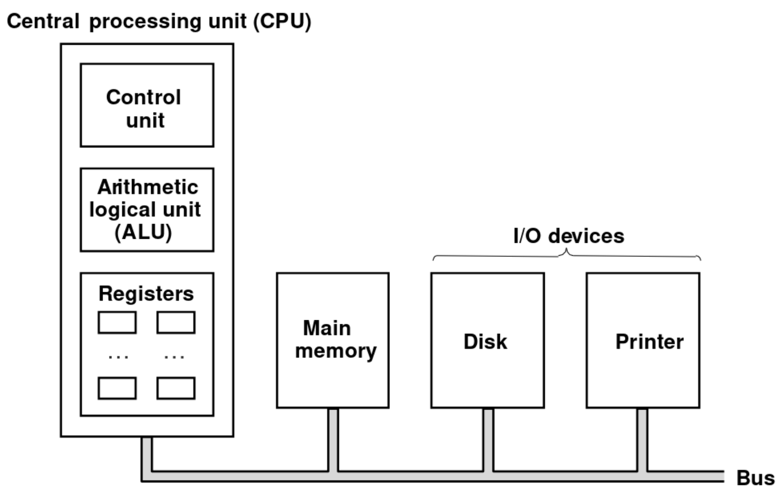
- **Hardware**: mechanical and electronic components
- **Software**: programs running on it
- **Firmware**: micro-programs stored in ROM



Note 1.1.0.1. Logical equivalence between HW and SW.

1.1.1 Micro-Computer

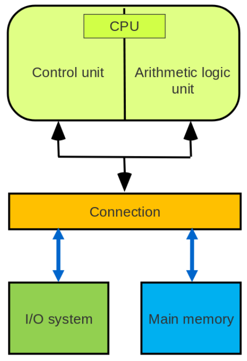
A micro-computer classical architecture is composed of different **modules**, each one dedicated to a feature (and to a corresponding physical part). They're all connected through a **bus**.



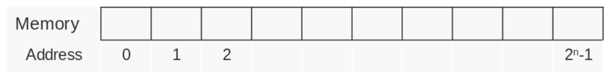
1.2 Architecture

1.2.1 Von Neumann

This architecture is one of the most wide spread. It has the following components:



- **Central Processing Unit**: controls the flow and the execution of all instructions. Consists of:
  - **Control Unit**: interprets the instructions and generates the control commands for other components
  - **Arithmetic Logical Unit**: executes the instructions, if needed with I/O and memory modules
- **Memory**: storage of **data** and **programs** as sequences of bits. It consists of memory cells with a fixed length, each one accessed individually with an address.

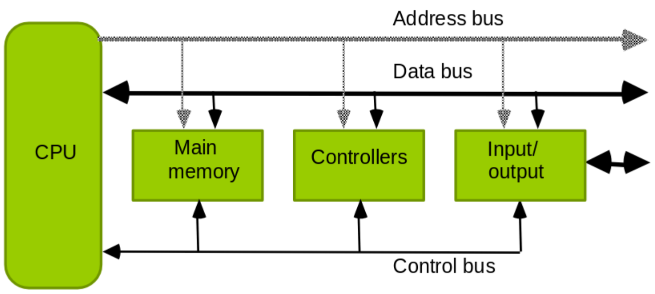


- **Input/Output** units: interface to the outside world that inputs and outputs the data

- **Interconnection**

  Observation 1.2.1 (Von Neumann Bottleneck)**.** Since there is no difference in memory between instructions and data and the CPU is way faster than the bus and the memory, the main interconnection is the central bottleneck.

IBM PC The IBM PC is a modified von Neumann architecture and was introduced by IBM fall 1981. The interconnection structure was realized by several busses.



Principle of operation This architecture follows the **Single Instruction Single Data** (SISD): at any time the CPU executes only a single instruction that can only manipulate a single operand.

There are **no memory protection** mechanisms, meaning that programs can destroy each other and access arbitrary data. Memory is **unstructured** and is addressed linearly. Interpretation of memory content depends on the context of the current program only.

Instruction processing follows a two phase principle:

- **Interpretation**: the content of a memory cell is fetched based on a program counter. This content is then interpreted as an instruction.
- **Execution**: the content of a memory cell is fetched based on the address contained in the instruction. This content is then interpreted as data.

Note 1.2.1.1. The instruction flow follows a strict **sequential** order.

Pros and cons The **advantages** of this architecture:

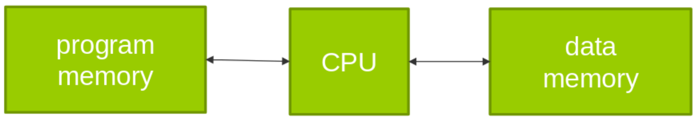
- Principle of minimal HW requirement
- Principle of minimal memory requirement

The **disadvantages**:

- Von Neumann bottleneck, also influencing programming approaches to deal with that
- Low structuring of data
- The instruction determines the operand type

1.2.2 Harvard

The main idea in this architecture is that there is a separation between data and program memory. Typically a processor uses this internally and then externally has a Von Neumann.



1.3 Performance

Definition 1.3.1 (Moore's Law)**.** *The number of transistors in an integrated circuit doubles about every two years.*

Observation 1.3.1 (Overclocking)**.** Given a certain processor with a clock speed, we can tune it up. The main consequence is that the energy used increases drastically, leading to higher and higher temperatures.

We need to assess the performance of a computer for different reasons: selecting a new one, tuning it o designing it. There are different methods to do so:

- Evaluation of **hardware parameters**, such as
  - Hypothetical maximum performance
  - **MIPS** o **MOPS**: millions of instructions/operations per second
  - **MFLOPS**: millions of floating point operations per second
  - **MLIPS**: millions of logical interferences per second
- **Run-time measurements** of existing programs through **benchmarks** that do different tasks (e.g. number crunching)
- **Monitoring** real cases of users through HW and SW monitoring
- Model **theoretic** approach with analytical methods or SW simulations

Benchmarks Here some example of benchmarks:

- **Sieve of Erathostenes**: uses the Ackermann function
- **Whetstone**: from 1970, uses FORTRAN with a lot of floating point arithmetic
- **Dhrystone**: from the begin of 1980s, focused on integers
- **Savage-Benchmark**: with mathematical standard functions
- **Basic Linear Algebra Subprograms** (BLAS), which is the core of the Linear Algebra Package
- **Lawrence Livermore Loops** for vectorizing compilers
- **SPEC**: Standard Performance Evaluation Corporation is a non profit organization created in 1989 that involves a lot of chip vendors. They focus on calculation speed and throughput.

1.4 Instruction Set Architecture

This type of architecture comprises the description of **attributes** and **functions** of a system from a machine language programmer's point of view. It enables the use of programs independent on the actual implementation. It defines:

- Instruction set
- Instruction formats
- Addressing modes
- Interrupt handling
- Logical address space
- Register and memory model accessible by the programmer

Note 1.4.0.1. It does not describe internal operations, components and implementation.

1.5 Micro Architecture

This type of architecture comprises the description of **hardware structure**, **data paths**, and **internal logic** of a specific realization of a processor. It defines:

- Number and stages of pipelines
- Usage of super scalar technology
- Number of ALUs
- Organization of cache memory

Definition 1.5.1 (Binary compatibility)**.** *All microprocessors following the same processor architecture specification are called* ***binary compatibles****.*

1.6 Classification

To classify different architectures we use the type of supported **parallelism**.
A parallel program can be seen as a partially ordered set of instructions, where the order is given by the dependencies among them. Independent instructions can be executed in parallel.
We have five levels of parallelism:

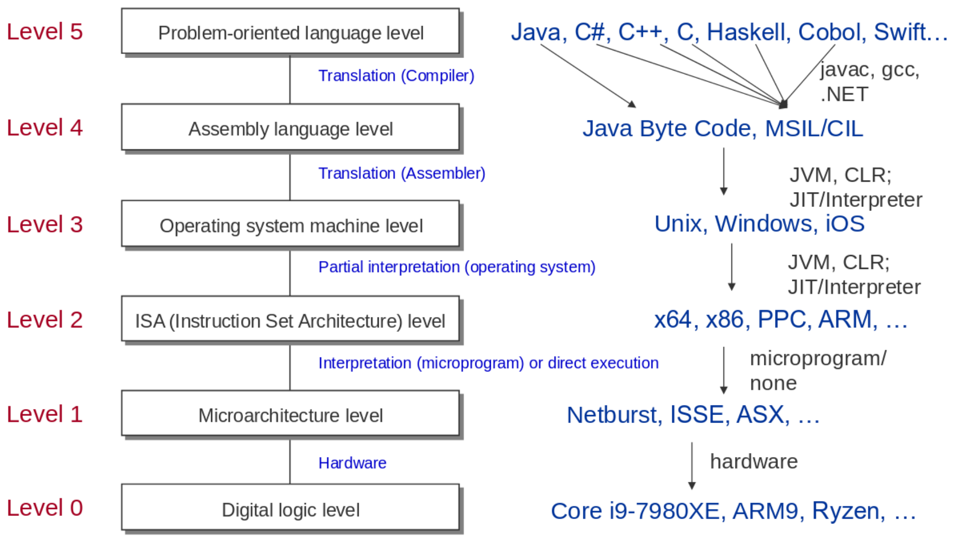
1. **Program**
2. **Process** or **task**: it involves heavy weighted processes or coarse-grained tasks
3. **Block**: it involves threads or light-weighted processes
4. **Instruction**: about the basic instructions of the chosen language or instruction set
5. **Sub-Operation**: basic instructions of a language or instruction set can be divided into sub-operations/micro operations by a compiler/the processor

Granularity The granularity depends on the relation of computation effort over communication or synchronization effort. Typically we have:

- **Large** grained: program, process, and thread level
- **Fine** grained: instruction and sub-operation

Note 1.6.0.1. Sometimes, the block or thread level is considered as **medium** grained.

1.7 6-Level model


Level 5
Problem-oriented language level
Translation (Compiler)
Level 4
Assembly language level
Translation (Assembler)
Level 3
Operating system machine level
Partial interpretation (operating system)
Level 2
ISA (Instruction Set Architecture) level
Interpretation (microprogram) or direct execution
Level 1
Microarchitecture level
Hardware
Level 0
Digital logic level
Java, C#, C++, C, Haskell, Cobol, Swift…
javac, gcc, .NET
Java Byte Code, MSIL/CIL
JVM, CLR; JIT/Interpreter
Unix, Windows, iOS
JVM, CLR; JIT/Interpreter
x64, x86, PPC, ARM, …
microprogram/ none
Netburst, ISSE, ASX, …
hardware
Core i9-7980XE, ARM9, Ryzen, …

2 Data arithmetic

2.1 Number systems

Definition 2.1.1 (Positional)**.** *Representation of numbers as a sequence of digits z_i, with the radix point between z_0 and z_{-1}:*

$$-z_n z_{n-1} \dots z_1 z_0 . z_{-1} z_{-2} \dots z_{-m} \tag{1}$$

Each position i of the sequence of digits is assigned a value, which is a power b_i of the base b of the numbering system:

$$X_b = z_n \cdot b^n + z_{n-1} \cdot b^{n-1} + \dots + z_1 \cdot b^1 + z_0 \cdot b^0 + z_{-1} \cdot b^{-1} + \dots + z_{-m} \cdot b^{-m} = \sum_{i=-m}^{n} z_i \cdot b^i \tag{2}$$

Common bases of numbering systems used in computer science are:

Base (b)	**Number system**	**Alphabet**
2	Binary	$0, 1$
8	Octal	$0, 1, 2, 3, 4, 5, 6, 7$
10	Decimal	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
16	Hexadecimal	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$

2.2 Conversion FROM decimal

2.2.1 Euclid

Considering the following representation of a number that we want to convert in a base b:

$$\begin{aligned} Z &= z_n \cdot 10^n + z_{n-1} \cdot 10^{n-1} + \dots + z_1 \cdot 10 + z_0 + z_{-1} \cdot 10^{-1} + \dots + z_{-m} \cdot 10^{-m} = \\ &= y_p \cdot b^p + y_{p-1} \cdot b^{p-1} + \dots + y_1 \cdot b + z_0 + y_{-1} \cdot b^{-1} + \dots + y_{-q} \cdot b^{-q} \end{aligned}$$

we generate the digits step by step starting with the most significant (leftmost):

1. Search p according to the inequation

$$b^p \leq Z < b^{p+1}$$

   and assign $i = p$ and $Z_i = Z$

2. Derive y_i and the reminder R_i by division of z_i by b^i

3. Repeat step 2 for $i = p - 1, \dots$ and replace each step Z_i with R_i until $R_i = 0$ or b^i is small enough that the precision is enough

2.2.2 Horner

This method is structured in two phases:

1. **Integer** part: factoring out the integer we get

$$X_b = \sum_{i=0}^{n} z_i \cdot b^i = ((\dots(((z_n \cdot b + z_{n-1}) \cdot b + z_{n-2}) \cdot b + z_{n-3}) \cdot b \dots) \cdot b + z_1) \cdot b + z_0$$

2. **Decimal** part: we multiply the decimals of the number by base b to get the fractional digits y_{-i} from the most to the least significant position

$$Y_b = \sum_{i=-m}^{-1} y_i \cdot b^i = ((\dots(((y_{-m} \cdot b^{-1} + y_{-m+1}) \cdot b^{-1} + y_{-m+2}) \cdot b^{-1} + y_{-m+3}) \cdot b^{-1} \dots) \cdot b^{-1} + y_{-1}) \cdot b^{-1}$$

2.3 Conversion TO decimal

We represent the values of the single positions of the number we want to convert in our common decimal system and sum all values.
The value X_b of the number is the sum of all single values of all positions $z_i \cdot b^i$, like in equation (2).

2.4 General conversion

If we want to convert from an arbitrary system to another, we first convert to decimal and then we use one of the other two methods.

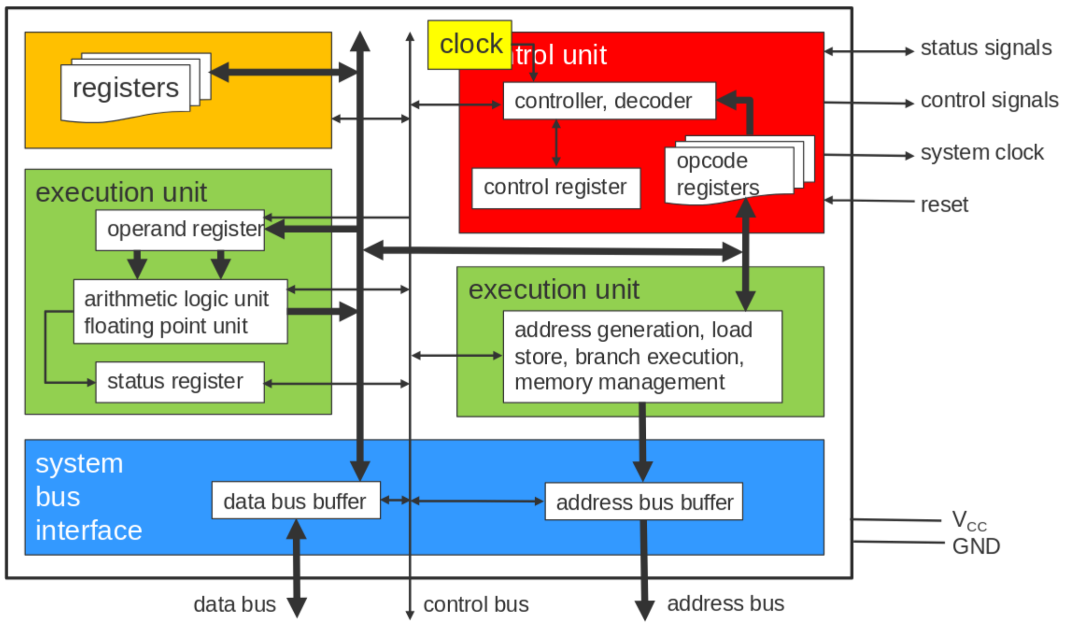
Observation 2.4.1. If the base of one system is a power of the base of another system the conversion is done by Replacing a sequence of digits by a single digit or replace a digit by *a* sequence of digits, respectively.

3 Microarchitecture

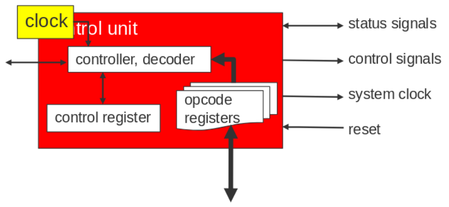
This section describes level no. 1 from the 6 level model that we saw in 1.7.

3.1 Architecture of a microprocessor

The internal architecture of a simple microprocessor is designed as follows:



3.1.1 Control unit



The control unit is the part that **controls** all the other components.

Clock Generates the system clock for distribution to all components.

Decoder It generates all control signals for the components and uses status signals and opcode as an input. Often **micro-programmable**.

Definition 3.1.1 (Micro programmable)**.** *The processor stores a* ***microprogram*** *for each instruction that can be modified only by the manufacturer. Single bits of a* ***micro instruction*** *represents* ***micro operations****, thus a setting of the control signals for the components.*
Pure ***RISC*** *processors typically use a fixed sequential circuit instead.*

The **phases of execution** of an instruction is composed of:

1. **Fetch**: load the next instruction into the opcode register
2. **Decode**: get the start address of the microprogram representing the instruction
3. **Execute**: the microprogram controls the instruction execution by sending the appropriate signals to the other components and evaluating the returned signals

Opcode register It contains the portion of the **instruction** that specifies the currently executed operation to be performed and eventual **opcodes**. It has **several registers** because:

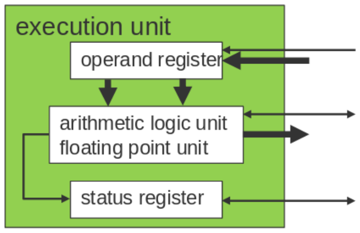
- Different instructions may have **different sizes**
- Opcode **prefetching** may speed up program execution: while decoding the current instruction the following one may be prefetched

Control register It stores the current status of the control unit. The meaning of the different bits depend on the specific processor. Some examples:

- *Interrupts* enable: determines if the processor reacts to interrupts
- *VM extension* enable: enable HW assisted virtualization on x86 CPUs
- *User mode instruction prevention*: if set, certain instructions cannot be executed in user level

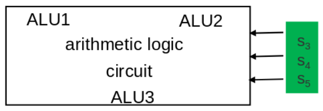
3.1.2 Execution unit

The execution unit executes all **logic** and **arithmetic operations** controlled by the control unit (e.g. int and float arithmetic, logic operations, address operations).



Communication with control unit Single bits of a micro instruction directly control the ALU operation and its operands.

Example 3.1.1. Example of micro instruction that sets S_3, S_4 and S_5:



S_3	S_4	S_5	ALU_3
0	0	0	$\text{ALU}_1 + \text{ALU}_2 + C_{in}$
0	0	1	$\text{ALU}_1 - \text{ALU}_2 - (\neg C_{in})$
0	1	0	$\text{ALU}_2 - \text{ALU}_1 - (\neg C_{in})$
0	1	1	$\text{ALU}_1 \vee \text{ALU}_2$
1	0	0	$\text{ALU}_1 \wedge \text{ALU}_2$
1	0	1	$(\neg \text{ALU}_1) \wedge \text{ALU}_2$
1	1	0	$\text{ALU}_1 \oplus \text{ALU}_2$
1	1	1	$\text{ALU}_1 \leftrightarrow \text{ALU}_2$

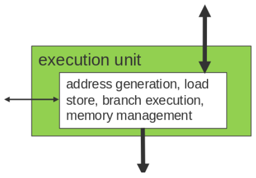
Status register Informs the control unit about the state of the processor after an operation via single bits. Common **flags** (the bits) are:

- **Auxiliary Carry** (AF): indicates a carry between the *nibbles* (4bit halves of a byte). It's used for **binary coded digit** arithmetic. [1]
- **Carry Flag** (CF): indicates a carry produced by the MSBs. Allows for additions and subtractions between **numbers larger** that a single word through sequential operations that take the carry into account.
- **Zero Flag** (ZF): if the result of an operation was zero. Used for **conditional branches**.
- **Even Flag** (EF): if the result is **even** or **odd** based on the LSB
- **Sign Flag** (SF): if the result is negative (MSB is 1) in two complements. Used for **conditional branches**.
- **Parity Flag** (PF): if the number of set bits is even or odd. Used for **error detection**.
- **Overflow Flag** (OF): if the result of an operation is too large to be represented

Program Status Word Status register combined with control register determine the current **state** of a processor. If we combine that with the **program counter** we get the state of a processor at a certain instruction of a program. It gets:

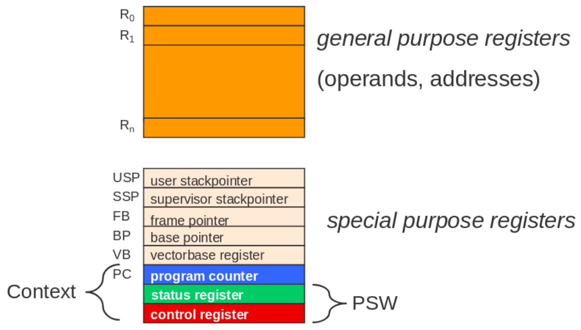
- **Pushed** to stack before context switch
- **Pulled** from stack to continue execution of an interrupted program

[1] Also called *half-carry flag*, *digit carry*, *decimal adjust flag*.


execution unit
address generation, load store, branch execution, memory management

Address generation unit Calculates the address based on control signals from the control unit and possible additional content of the registers.

3.1.3 Registers

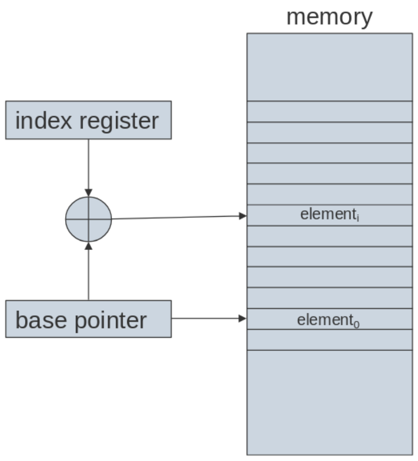


They extend the registers of the execution unit, containing frequently used operands. It's a very **fast** memory with a low access time ($<$ 1ms). The **selection** of single registers happens through dedicated **control lines** without any address decoding. It can also offer other functions:

- **Increment** and **decrement**
- **Shift**
- Set to **zero**

There are several independent I/O ports, allowing **simultaneous** writing and reading of different registers. In modern processors up to 4 writes and 8 reads in one clock cycle.

Base pointer The base pointer special register contains the start address of a memory region (e.g. an array). The **index** represents the offset relative to the base pointer. The sum of the two gives the absolute address of an element.



Index register The **index register** has special functions to save time from the ALU:

- **Post-increment**: automatic increment of the register by n after addressing the memory
- **Pre-decrement**: automatic decrement of the register by n before addressing the memory
- **Auto-increment/decrement**
- **Autoscaling** by a factor n, used to access memory in bytes or words

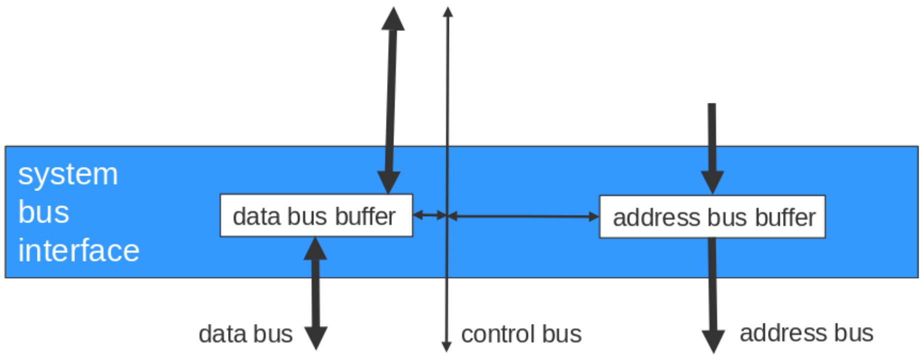
Stack pointer Part of the main memory organized with **LIFO** principle. It stores the **PSW** during the subroutine calls/interrupts, allows the passing of the **parameters** and stores **temporary results**. Processors usually have different types (e.g. user, system, data).
The **stack pointer** contains the address of the newest data on the stack.
Two operations are allowed:

- **PUSH**: transfer the value of a register on top of the stack
- **POP**: load a register with the value on top of the stack and remove it

3.1.4 System Bus Interface

The Bus Interface Unit is the connection of the microprocessor to all the other components of the computer. The main purposes are:

- **Buffering** of *addresses* and *data* (operand and instructions)
- **Adaptation** of *clock cycle*, *bus width* and *voltages*
- **Tristate**: detaches the processor from the external bus



3.2 Pipelining

The performance of a computer can be improved either by improving the **hardware** using new technologies and materials (expensive and with limitations) or by increasing the **parallelism** (more transistors, wider bus, replicated function units).

3.2.1 Structural enhancements

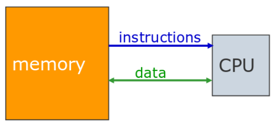
The classification of computer architectures, according to Flynn, is composed of four types.

SISD Single Instruction Single Data consists in one single stream of instructions that operate on the data (classic Von Neumann).

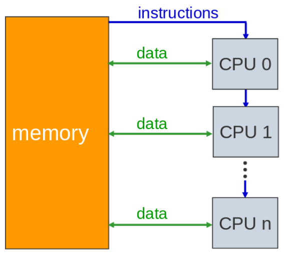
SIMD Single Instruction Multiple Data is where all processors perform the same instruction on different data (array processor). E.g. image processing, where each processor handle one part of the image.

MIMD Multiple Instruction Multiple Data where all processors perform different instructions on different data. The one it's mostly used today.

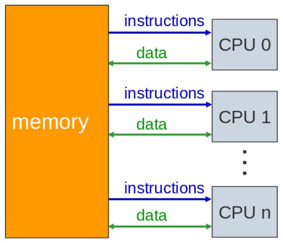
MISD Multiple Instruction Single Data where several instructions operate on a single data. Not very common but could be use e.g. for fault tolerance (and then compare the results).



(a) SISD



(b) SIMD



(c) MIMD

3.2.2 Pipeline processing

Pipelining is the **subdivision** of an operation into several phases or sub operations and their **synchronous execution** in different functional unit (each one responsible for its own).
Each stage is a pipeline **stage** or **segment**. The whole pipeline is clocked in a way that each cycle instruction can be shifted one step further through it.
Ideally we have a **k stage pipeline** with k cycles and k stages, allowing k instructions to be executed at the same time and needing k cycles to complete.

Definition 3.2.1 (Latency)**.** *Duration of the complete processing of an instruction. The time an instruction needs to go through all k stages of the pipeline.*

Definition 3.2.2 (Throughput)**.** *Number of instructions leaving the pipeline per clock cycle. It should be close to 1 for a **scalar** processor and may be >1 for **superscalar** processors.*

Speedup A **standard** processor without pipeline, considering n instructions and k stages, needs:

$$n \cdot k \text{ cycles} \tag{3}$$

A **pipelined** processor needs, assuming k cycles latency and throughput $= 1$, needs:

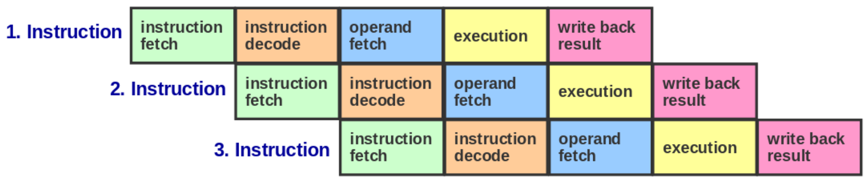
$$k + (n-1) \text{ cycles} \tag{4}$$

Assuming an infinite number of instructions the **speed up** of a processor with a k staged pipeline is:

$$\lim_{n\to\infty} \frac{n \cdot k}{k+n-1} = \lim_{n\to\infty} \frac{k}{\frac{k}{n}+1-\frac{1}{n}} = \frac{k}{0+1-0} = k \tag{5}$$

Five stage pipeline It consists in these five stages:

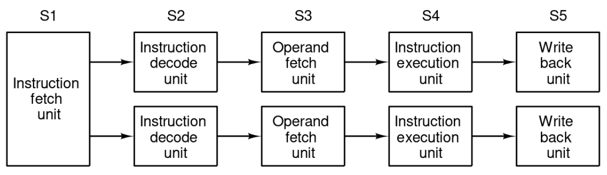
1. **Instruction Fetch** (IF): load the instruction from memory or cache into the opcode register and increment the program counter
2. **Instruction Decode** (ID): generate internal signals based on the opcode or jump to the appropriate microprogram
3. **Operand Fetch** (OF): load the operands from the registers into the operand registers of the ALU and calculate the effective address for load/store or branch instructions
4. **Execution** (EXE)
5. **Write Back** (WB): write the result into a register or memory if needed. Load/store instructions put the address on the address bus and transfer the data to the memory



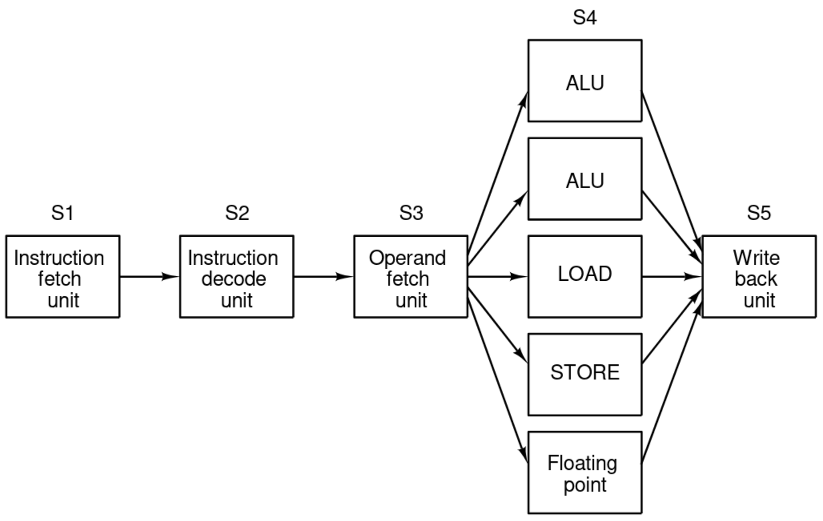
Two instruction If we assume that:

- Instruction Fetch requires half the time of all other stages
- We can have arbitrary many other functional units

then we can have an improved pipeline that works in parallel:



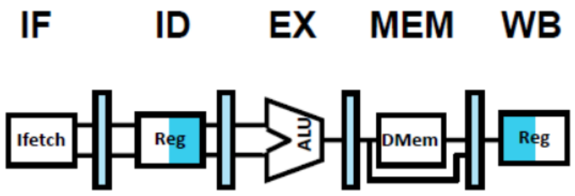
Specialized execution units



MIPS pipeline The MIPS pipeline is defined as follows:

1. **Instruction Fetch** (IF): fetch instruction from *main* memory
2. **Instruction Decode** (ID): read operand registers while decoding the instruction (simultaneous thanks to format of MIPS instructions)
3. **Execution** (EXE): **execute** the operation (arithmetical and logical operations) or **calculate** an address (*load* and *store*)
4. **Memory Access** (MEM): access an operand in data memory[2]
5. **Write Back** (WB): write the result into a register

With this pipeline the affected registers are **written** into during the first half of the WB stage while the operand registers are **read** during the second half of the ID stage.



3.2.3 Pipeline hazards

Definition 3.2.3 (Pipeline hazard). *Phenomena that may disrupt the smooth execution of a pipeline.*

Example 3.2.1. If we assume a unified cache with a single read port (instead of separate instruction and data caches), a memory read conflict appears among IF and OF stages. The pipeline has to stall (**pipeline bubble**) one of the accesses until the required memory port is available.

[2]If the instruction is *load* or *store*, otherwise nothing happens.

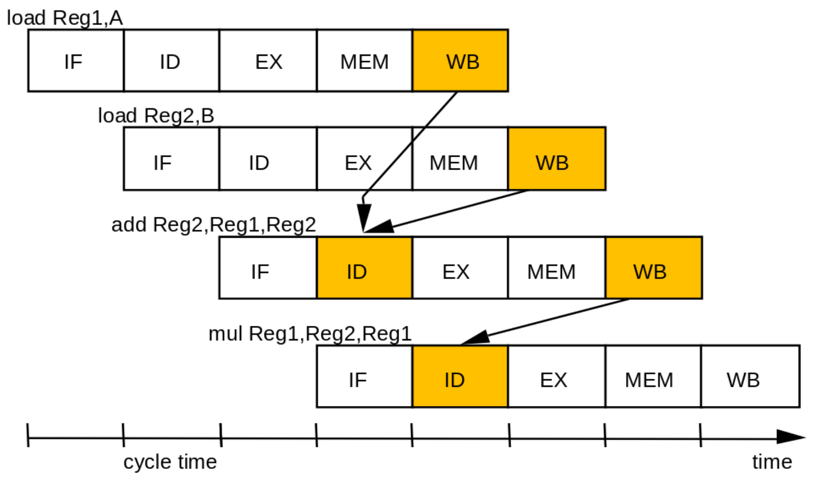
Data hazard These type of hazards happen when there are **dependencies** between instructions, causing an overlapping between their execution. There are three types:

- **Read After Write** (RAW): instruction 2 tries to read the register before instruction 1 writes it
- **Write After Read** (WAR): instruction 2 tries to write a register before instruction 1 reads it
- **Write After Write** (WAW): instruction 2 writes a register before instruction 1 writes it

Example 3.2.2 (RAW in MIPE pipeline)**.** Let's consider this piece of code:

```
mov R1, A
mov R2, B
add R2, R1, R2
mul R1, R2, R1
```

We will have the following dependencies:



Observation 3.2.1 (WAR and WAW)**.** In a five stage pipeline WAR and WAW can't happen since all instructions take five stages and the reads are always in stage 2 while the writes in stage 5.

Software solutions Software solutions involve two options:

- Put **no-op** operations after each instruction that may cause a hazard
- **Reorder** the instructions

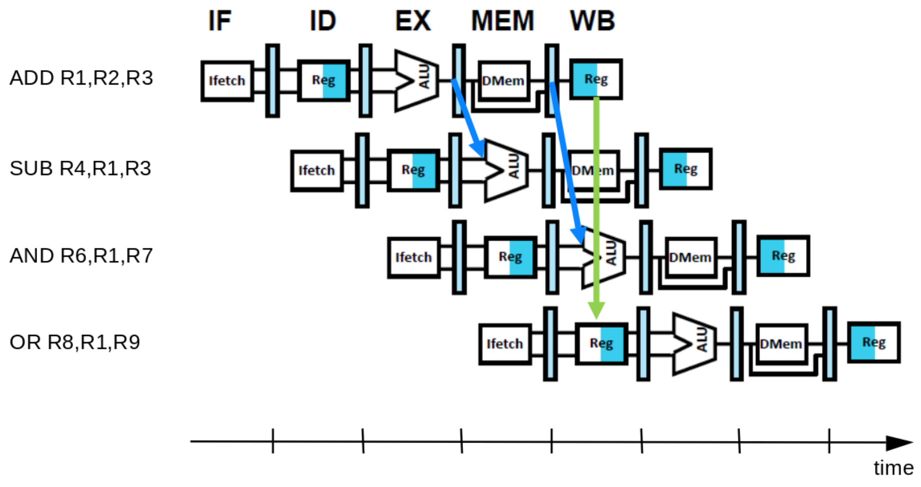
Hardware solutions Hardware solutions require a hazard detection logic:

- **Stalling** or **interlocking**: we stall the pipeline for one or more cycles
- **Forwarding**: we forward either the **result** or the **load**ed data directly to the next stage where it's needed. It may be necessary to also use **stalling** since even if forwarded the data may still not be available.

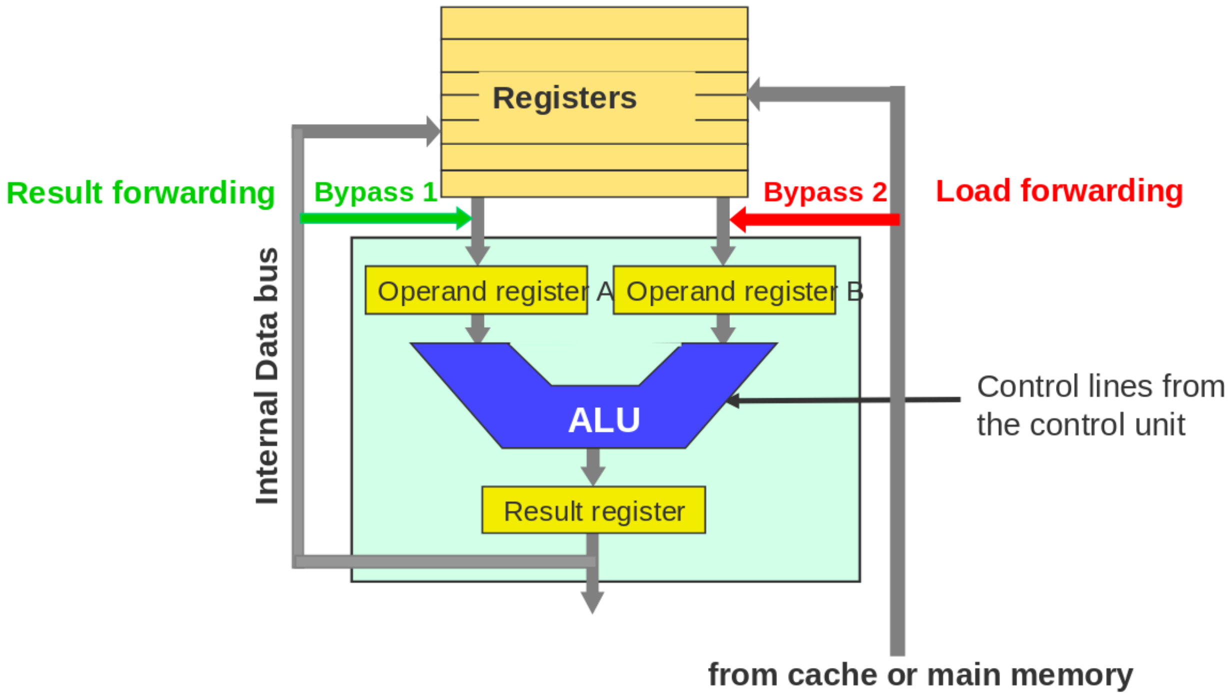
Example 3.2.3 (Forwarding)**.** Let's consider this piece of code:

```
add R1, R2, R3
sub R4, R1, R3
and R6, R1, R7
or R8, R1, R9
```

We will have the following forwarding of both, result and load



This is how a hardware (**bypass**) solution can be implemented:



Structural hazard Structural hazards may arise from some combinations of instructions that cannot be accommodated because of **resource conflicts**.

Example 3.2.4. If the processor has only one register port and two instructions want to write into it at the same time.

The main solutions are:

- **Arbitration with interlocking**: hardware that performs resource conflict arbitration and interlocks one of the competing instructions
- **Resource replication**: e.g. more write ports. This could arise more dependencies.

Control hazard Control hazards arise from **control flow instructions** (e.g. branch, jump).

Example 3.2.5. Considering this code:

```
           cmp R1,R2
           adc R4,R5,R4
           beq Label
           add R3,R1,R2
Label:     sub R6,R4,R5
           sll R0
```

The *add* instruction is still in the pipeline when the jump is executed and therefore it is also executed before it.

Software solution There are two options:

- Put **no-op** operations after every branch
- **Reorder** the instructions so that the ones that will not affect the branch will be executed right before it

Hardware solution There are three options:

- **Interlocking**: the hardware detects the branch and applies the interlocking to stall the next instructions
- **Flushing**: empty the pipeline before releasing the jump
- **Speculative branch**: in case of a **conditional jump** estimate the result and load the pipeline. If the estimate was wrong, *flush*.

3.2.4 Branch prediction

Branch prediction foretells the outcome of conditional branch instructions. IF stage finds a branch instruction and predicts its direction. The branch delay slots are speculatively filled with instruction, either those of:

- the consecutively **following path**,
- the path at the **target address**

After resolving of the branch direction decide upon correctness of prediction. In case of misprediction discard wrongly fetched instructions. Re-rolling in this case is **expensive**.

Branch Target Buffer The Branch Target Buffer (BTB) or Branch-Target Address Cache (BTAC) stores branch and jump target addresses. The BTB is accessed during the IF stage and consists of a table with **branch addresses**, the corresponding **target addresses**, and **prediction information**. There are some variations:

- **Branch Target Cache** (BTC): stores one or more target instructions additionally
- **Return Address Stack** (RAS): a small stack of return addresses for procedure calls and returns is used additional to and independent of a BTB.

Static The prediction for an individual branch does not change and comprises of:

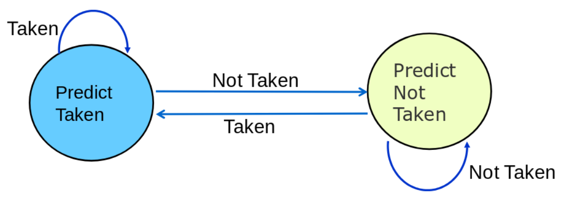
- **machine-fixed**: the prediction can be **wired** into the processor by predicting that all branches will be taken (backwards) or all not taken (forward)
- **compiler-based**: opcode in branch instructions allows the compiler to set it. Then it can either **examine** the program code or use **profile information** from earlier runs

Dynamic In dynamic branch prediction the prediction is decided on the **computation history** of the program execution. In general gives better results than static but at the cost of **increased hardware complexity**.

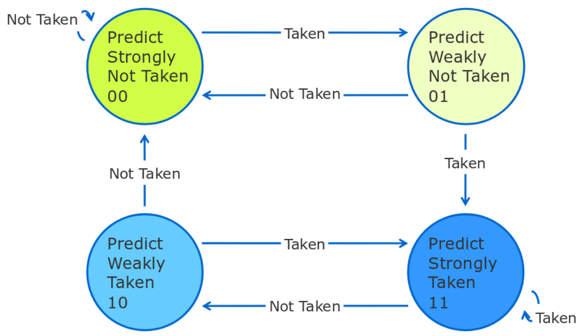
Predictors A **one-bit** predictor correctly predicts a branch at the end of a loop iteration, as long as the loop does not exit. In nested loops, a one-bit prediction scheme will cause two mispredictions for the inner loop:

- One at the **end** of the loop, when the iteration exits the loop instead of looping again
- One when executing the **first loop iteration**, when it predicts exit instead of looping

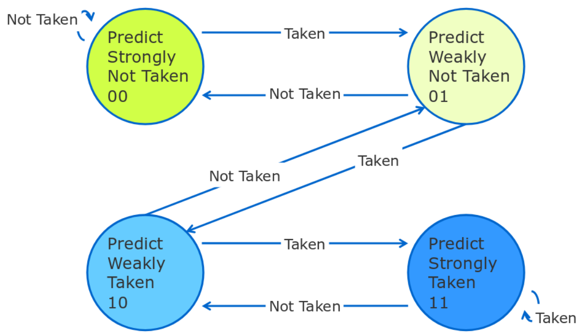
This is avoided by a **two-bit** predictor scheme, where a prediction must miss twice before it is changed when a two-bit prediction scheme is applied.



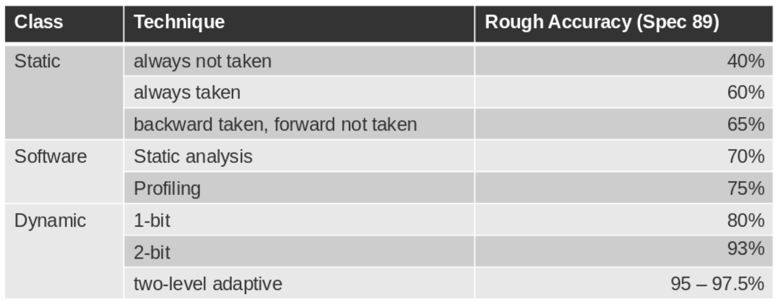
(d) one-bit


Not Taken
Predict Strongly Not Taken 00
Taken
Not Taken
Predict Weakly Not Taken 01
Not Taken
Taken
Predict Weakly Taken 10
Taken
Not Taken
Predict Strongly Taken 11
Taken

(e) two-bit Hysteresis scheme


Not Taken
Predict Strongly Not Taken 00
Taken
Not Taken
Predict Weakly Not Taken 01
Not Taken
Taken
Predict Weakly Taken 10
Taken
Not Taken
Predict Strongly Taken 11
Taken

(f) two-bit Saturation Counter scheme



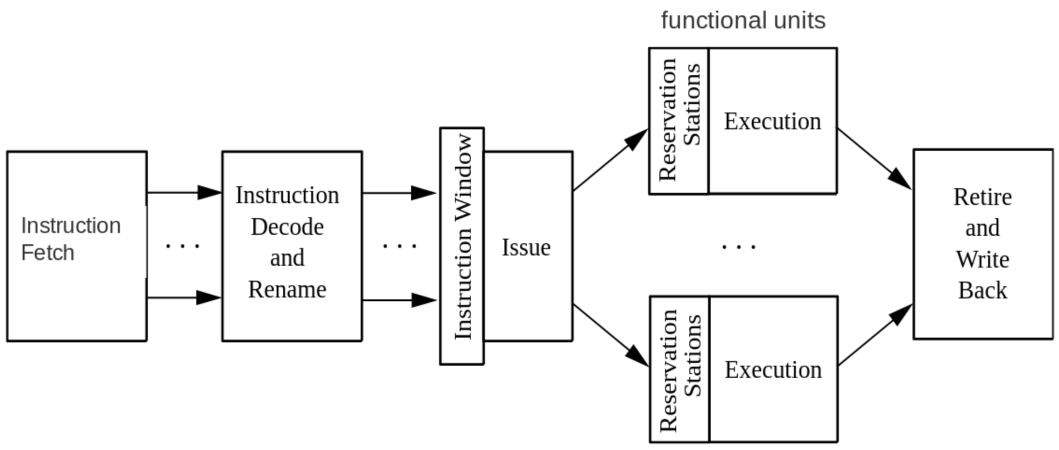
Class	Technique	Rough Accuracy (Spec 89)
Static	always not taken	40%
	always taken	60%
	backward taken, forward not taken	65%
Software	Static analysis	70%
	Profiling	75%
Dynamic	1-bit	80%
	2-bit	93%
	two-level adaptive	95 – 97.5%

Figure 1: Statistics about different techniques

Predicate instructions It provides **predicated** or **conditional instructions** and **predicate registers**, which are used as an additional input operand. They are able to remove a branch and keep the pipeline busy but it complicates the instruction set and consume resources.

3.3 Superscalar processors

Definition 3.3.1. *Superscalar machines are distinguished by their ability to dynamically issue multiple instructions each clock cycle from a conventional linear instruction stream.*



A superscalar pipeline is divided in three sections:

- **in-order**
  - *fetch*: get the commands
  - *decode*: decode the new instructions
  - *rename*: externally visible register are mapped to internal shadow registers, stored in a rename map (e.g. *alias table*), so that core execution units are free from name dependencies
  - *issue* if *in-order* issue processor. The waiting instructions in the **waiting window** are examined and then simultaneously (up to the *issue bandwidth*) **issued** to the **functional units**
- **out-of-order**
  - *issue* if out-of-order issue processor
  - *execution*: there is a **reservation station**, a buffer that contains a single instruction operands. There may be one for multiple units. When all the operands are ready the instruction is **dispatched**.
  - *completion*: when the result is ready for forwarding and buffering. The reservation station gets freed and the state of the execution (can be an interrupt) is noted in the **reorder buffer**
  - *Commitment*: after completion an instruction gets committed if:
    - ∗ al previous instructions are committed or can be committed in the same cycle
    - ∗ if no interrupt occurred before and during the execution
    - ∗ if the instruction is not speculative anymore
- **in-order**
  - *retirement*: an instruction retires when the reorder buffer is freed either because of a commit or a delete
  - *write-back*: the result of the instruction is made permanent in the architectural register

4 Instruction set architecture

The ISA level is the classical boundary between SW and HW and defines what is visible or not to the user. High level languages are translated into instructions from the ISA.

4.1 CISC

Complex Instruction Set Computer exist for chips, high programming languages and special purpose applications that favor complex instructions.
The **positive sides** are:

- Execution of a single complex instruction is **faster** than the execution of a microprogram that does the same thing
- Shorter programs, thus **faster loading times**
- Direct **support of programming constructs** from higher programming languages
- Support of specialized **powerful compilers**
- **Compatibility**
- Support of **special purpose applications** (e.g. matrix operations)

The **negative sides**:

- Needs faster main memories and the use of cache memory speed-up program execution
- Complex instructions can sometimes be **replaced with** several **simpler** and faster ones
- **Long development** cycle of CPUs
- **Complex control units**
- Large microprograms with potentially **more errors**
- Real programs often use only a small portion of the instructions

While there are many powerful instructions, usually just 20% of them are used, still maintaining the complex format. Furthermore, many classical CISC architectures have a CPI > 2.
That being said, optimized code for Pentium, Itanium and others usually have a CPI ≈ 1.

4.2 RISC

Reduced Instruction Set Computer consists of:

- Few **instructions** (≤ 128)
- ≤ 4 **instruction formats**
- Fixed **instruction length** of 32 bit
- ≤ 4 **addressing modes**

If possible all instructions should be implemented so that they can finish in a single cycle. As a consequence, there is no micro programming in RISC.

Note 4.2.0.1. Early RISC processors had SW controlled pipeline (NOPs) instead of special hardware.

4.2.1 Memory access

This instruction set uses the **register/register** or **load/store** principle: **memory access** is possible only via *load* and *store* operations, while all other operations are done on registers.

4.2.2 Pros and cons

The **positive sides** are:

- Single chip implementation, allowing the use of the saved **space** for something else
- Short development cycle
- **Higher clock rates**, **pipelines**

The main **downside** is the bottleneck in the memory interface, since main memory is much slower than internal registers and cache.

4.3 Addressing modes

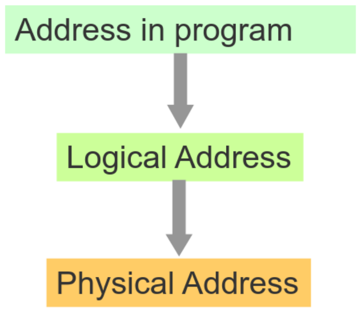
There are different possibilities to calculate the address of an operand or the branch target address in the memory.

4.3.1 Static

The **classical** way is through an **absolute** address. This means that the location is determined at compilation and accessing dynamic structures requires a change in the address for each instruction.

4.3.2 Dynamic

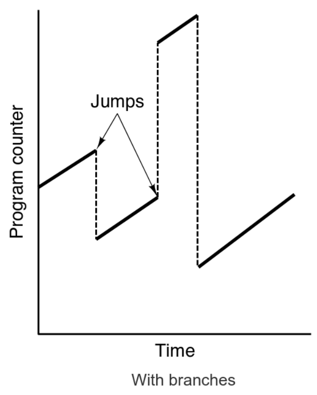
In this case the calculation of the address is done at runtime: the instruction triggers the calculation of the **logical address** and then the memory management unit finds the physical one.

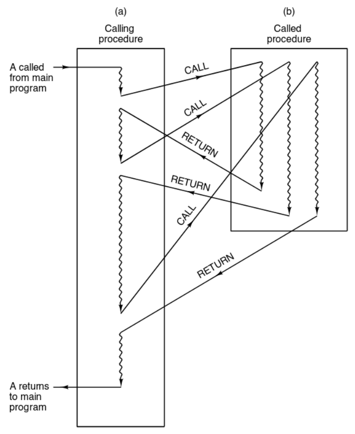


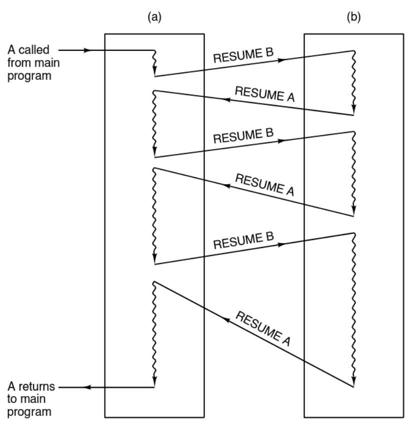
4.4 Non linear execution

Usually a program follows a non linear execution due to many reasons:

- Jumps, branches
- Procedure calls, subroutines, method calls
- Multi-threading, parallel processing, co-routines
- Hardware interrupts
- Traps, software interrupts







Definition 4.4.1 (Exception)**.** *Interruption of the programmed control flow of instructions during runtime.*

4.4.1 Causes

Exceptions may be caused by:

- **External** reasons: asynchronous events such as
  - *RESET*: reset of the processor (e.g. power button)
  - *HALT*: stop the execution of the processor (e.g. to avoid conflict on memory access)
  - *ERROR*: call of an error routine (e.g. bus errors)
  - **Interrupts**: request triggered by an external device (e.g. to announce incoming data). They can be *maskable* or *non maskable*
- **Internal** reasons: synchronous events such as
  - **Software interrupts**: *INT* instructions in the program triggers an interrupt (e.g. system call)
  - **Traps**: exception caused by internal events (e.g. overflow)

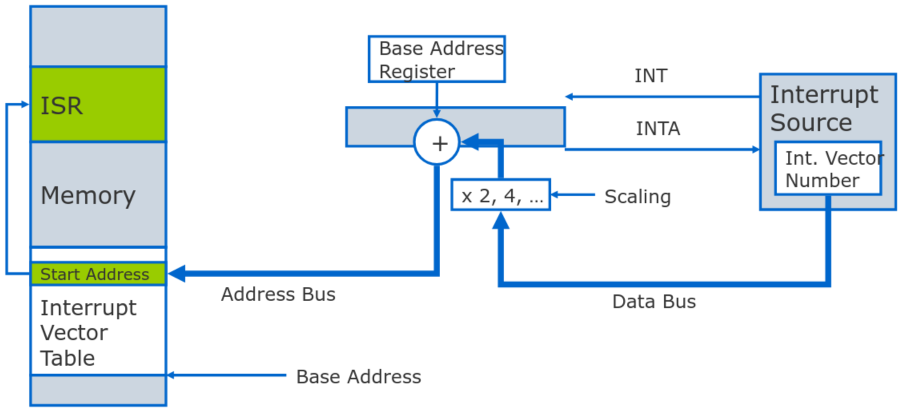
4.4.2 Handling

Handling of exceptions requires specialized routines called **Interrupt Service Routine** (ISR). They have the same structure as a subprogram but with some differences:

Activity	**Subprogram**	**ISR**
Activation	*call* subroutine	*INT* instruction or hardware activation
Return after completion	*RET* instruction	*RETI* instruction
Calculation of starting address	Written in calling program	Determined via interrupt table
Saving status	Typically saves PC on a stack	Saves PC and PSW on a stack

The **steps** of an ISR are:

1. Interrupt activation
2. Finalize the current instruction
3. Check if SW or internal/external HW
4. Check if *Interrupt Enable* bit is set
5. If HW, find the source and activate the acknowledge (INTA)
6. Reset *Interrupt Enable* to avoid additional interrupts
7. Save PSW and PC on stack
8. Calculate the start address and load it to PC. Usually done through the **Interrupt Vector Table**, which is at a specific location and contains the start addresses of the ISRs

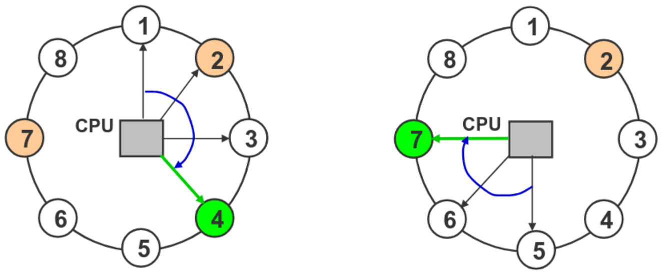


9. Execute the ISR:
   - Push the used register on stack
   - Set the interrupt bit to allow more interrupts
   - Execute ISR
   - Pop the registers from stack
   - Return using *IRET*
10. Restore PSW and PC and continue the original program

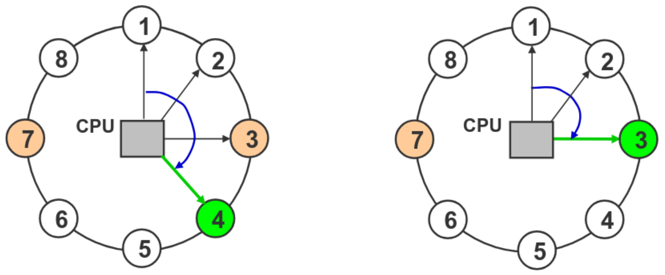
4.4.3 Multiple interrupts

During the handling of an interrupt, more may happen. There are two different approaches to handle them:

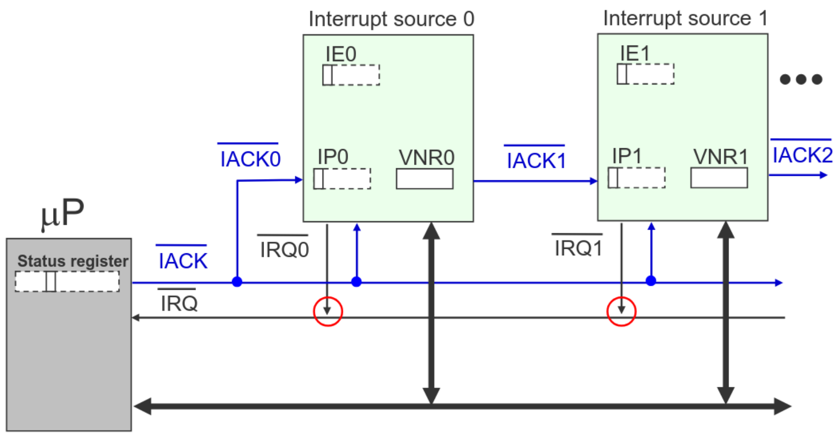
- **Fair**: continue cyclic polling following the last served source, granting an equal chance of being served



- **Priority**: cyclic polling always starts at a predetermined sourced. Every source gets a different priority and the higher one are served first.



It's possible to use **hardware daisy chaining** to handle multiple interrupts, using specialized hardware for prioritization and identification of interrupts. Each source for an interrupt uses dedicated HW for connecting with a successor and a predecessor. The first one has the highest priority.


Interrupt source 0
Interrupt source 1
IE0
IE1
$\overline{\text{IACK0}}$
IP0
VNR0
$\overline{\text{IACK1}}$
IP1
VNR1
$\overline{\text{IACK2}}$
µP
Status register
$\overline{\text{IACK}}$
$\overline{\text{IRQ0}}$
$\overline{\text{IRQ1}}$
$\overline{\text{IRQ}}$