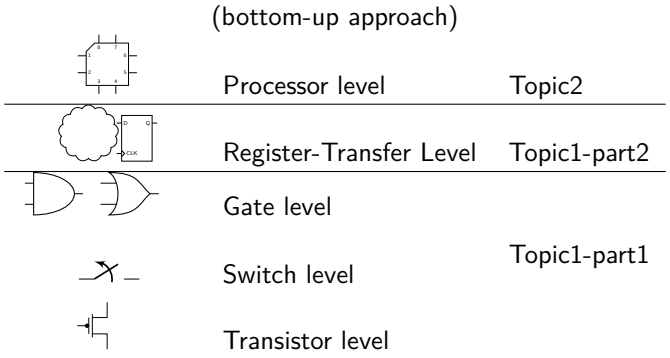
Lecture Notes on Essentials of Computer Systems - Topic 2, part 1

Table of Contents

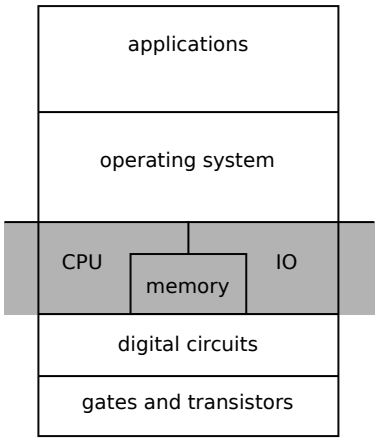
1 Introduction to Computer Architecture

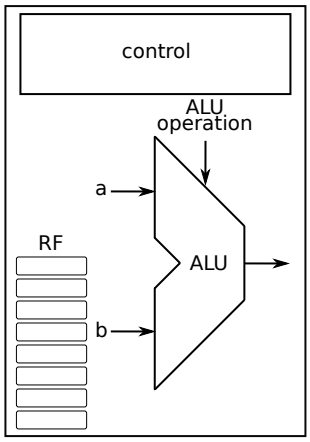
1.1 Levels of abstraction - part III.

We are following a bottom-up approach. Topic 1 covered logic gates and Boolean algebra (part 1), followed by RTL datapath design (part 2). We now begin discussing the processor level (Topic 2).



The left figure below shows the processor level (shaded grey) in more detail. The main elements we will discuss in this section are the Central Processing Unit (CPU), memory, and Input/Output System[1] (IO). We will focus on a simple general purpose[2] CPU, highlight important design principles, and show connections to the lessons learned in Topic 1. Then we will make a bridge to Topic 3, which will discuss operating systems and how they interact with the CPU.





The figure on the top right shows our first, still abstracted, view of the CPU. The main components of the CPU are the Arithmetic-Logic Unit (ALU), the register file (RF or RegFile) and the control unit (CU). The ALU is a combinational circuit with two n-bit data inputs (shown as a and b) and performs basic operations, for example addition, subtraction, logic operations, ..., and produces an n-bit output. The register file is an array of n-bit registers that provide the inputs to the ALU and hold (intermediate) results. The grouping n-bits is also referred to as "word" and is considered as a basic unit of computation. The number of words in the RegFile varies from system to system, but we often encounter Terminology 32-bit or 64-bit CPU refers to the parameter n, that is to the length of the word. We will work with $n = 32$, and with a RegFile that contains 32 registers[3]. The third component is the control circuitry: it sets the control signals to select the desired ALU operation, appropriate registers for its inputs and outputs, ... But how does the control unit know what to do ? The answer to this question is by decoding the instructions (we will use abbreviation *insn* for instruction).
The CPU functionality can be explained with three[4] main steps:

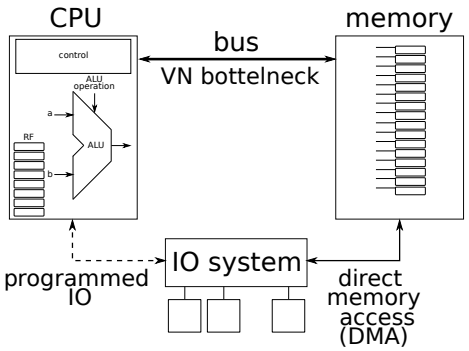
- fetch the insn
- execute the insn
- write back results

For now we take an insn as a fact and will go into more details in the next two sections.

[1] we will only briefly mention the IO system in this course
[2] opposed to domain specific co-processors, accelerators, e.g., dedicated to operations needed for graphics or signal processing
[3] in total 32×32 bits
[4] recall the abstraction we used for logic gates and basic building blocks in Topic 1: sample inputs, compute results, drive the output

1.2 Von Neumann model

The Von Neumann (computer) model is shown in the figure on the right. It consists of three major components: the CPU, the memory, and the IO system. The operation of the Van Neumann model is *completely determined by the sequence of instructions*: the CPU fetches the instructions from the (main) memory and executes them. For now we will view the memory as an array of memory words and each word is uniquely determined by an address. The memory word has the same meaning as the CPU and register word, but the two words may be of different lengths. An important question is which instruction to fetch next ? The CPU has a special register called Program Counter (PC), which holds the address of the insn.

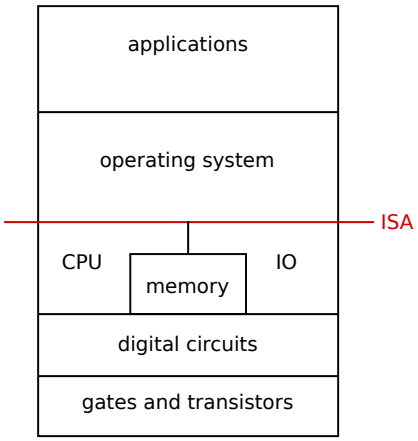


The bus between the CPU and the memory carries the following: the address, the data, and control (direction: read or write). The CPU must fetch both the insns and the operands from the memory, but since the bus is shared, it can not fetch both at the same time. This limits the system performance, and the memory bus is often referred to as the Von Neumann bottleneck. We will discuss ways to mitigate this problem in Chapter 3.

We will only briefly mention the Input/Output (IO) system. The IO devices[5] are not crucial for the functioning of the computer, rather they perform other tasks, such as interaction with the user (keyboard, monitor) and communication (network card). The hard drive is a bit of an exception: it is controlled by the IO system, but it is also a part of the memory hierarchy, which will be discussed in Chapter 3.

1.3 Instruction Set Architecture - ISA

In this section we will finally answer the question what is an instruction. We begin this discussion with the notion of the Instruction Set Architecture (ISA). The ISA is an abstract model that serves as the interface between hardware and software in computer systems. The figure on the right shows where the ISA fits in the levels on abstraction. *The ISA defines the set of instructions that a processor can execute.* It defines the number and type of instructions, functional behavior, encoding of the instructions, CPU and RegFile word length, programmer-visible registers[a][b], operations, operands, memory access, ... It represents the conceptual contract between the programmer and the machine, outlining the operations that a processor can perform and the encoding for each instruction. ISA encompasses the assembly language syntax and the underlying machine code, acting as a bridge that allows software developers to write programs without delving into the intricate details of hardware implementation. The architecture defines the capabilities and limitations of a processor, influencing its performance, power efficiency, and overall functionality.



[a] RegFile provides internal storage locations

[b] most CPU have a small set of programmer-visible registers and a larger number of registers used internally, for example to optimize the performance

Different computer architectures, such as MIPS, x86, ARM, and RISC-V, have distinct ISAs, each tailored to specific design philosophies and application domains. The evolution of ISA has played a pivotal role in shaping the landscape of computing, influencing advancements in hardware design, compiler optimization, and software development practices. It is very important to separate the ISA and the microarchitecture: the microarchitecture refers to a particular physical implementation of the CPU and different chips can implement the same ISA. They often feature different design decisions and trade-offs, for example low-power vs. high performance CPU. The ISA helps to hide these details. When developing software using a high-level programming language, the computer programmers is not aware of how hardware is designed and organized. At the same time, the computer designer (who develops the hardware) is not aware of high-level programming languages. In between them, there is a layer, which is the ISA. This layer "talks" to both entities: the hardware and the software. Therefore, is a software is converted into instructions following ISA definitions and the hardware is also developed according to this same ISA definitions, then software and hardware become compatible.

First, let us illustrate that different ISAs have different definitions and instructions. The most famous ISAs are MIPS, Intel

[5] sometimes called peripherals

x86, ARM and RISC-V. When the same software is compiled to each of these ISAs, the structure may be very different. The ISA removes this detail from a software developer using a high-level programming language, and futhermore, allows the same high-level program to be compiled for different ISAs.

MIPS	Intel x86	RISC-V	ARMv7
lw $t0, a	mov eax, [a]	lw $t0, a	ldr r0, a
lw $t1, b	mov ebx, [b]	lw $t1, b	ldr r1, b
add, $t2, $t1, $t0	add, eax, ebx	add, $t2, $t1, $t0	add, r2, r1, r0

Next we explain the the first example of an instruction in more detail:

machine code	assembly	our notation	meaning	CPU operation
000000 01000 01001 01010 00000 100000	add $t2, $t0, $t1	add R10, R8, R9	$c = a + b$	$R10 \leftarrow R8 + R9$

The first column shows the machine code, that is the insn encoded as 32 bits. This value is fetched by the CPU from the memory and executed. The second column shows the actual assembly used for MIPS32 architectures, and the third column shows our notation: we abstract away the details of register usage. The reason for this is that different ISAs dedicate the registers differently, as was shown in the example of different ISAs. Register R0 is usually reserved to always[6] hold the constant 0. The registers will hold the values of variables a,b,c. The CPU action column that the control unit will transfer b and c from registers R8 and R10 onto the ALU inputs, set the ALU operation (ALUop) to addition, and then store the sum c into register R10. We will use $\leftarrow$ to indicate where the r.h.s. can be found when the insn is completed. In example above we used actual register R10 as destination, but the ISA specifies them differently, for example it will list Rd. We will freely use registers R1 - R20 without any restrictions[7]. A quick note: a small RegFile is faster because larger number of registers increases their implementation area, not only for the registers but also the number of wires and select-logic needed to access them, which will in turn also increase the clock period.

There are a few ISA design principles, which we will only briefly mention. First is *simplicity favors regularity*. Regularity in the sense that most insns have 3 operands, 2 inputs and 1 output. As we have seen[8] in Topic 1, we want to keep the hardware as simple as possible, minimizing the need for extra elements, such as MUXes. Another important design principle, which we will revisit in the performance section, is *make the common case fast*. The use of constants is very common, hence we create dedicated insns, for example addition with a constant. With this example we also want to highlight another fact: *the number of ALU operations is much smaller then number of instructions*. For example $c \leftarrow a + b$ and $c \leftarrow a + 5$ both involve the addition operation (ALUop), but the first insn will have two register inputs while the second insn will have one register input and one constant input. We show the example using the constant (transferred into the CPU as the Immediate) in the table below. Encoding the constant as a part of the insn saves the time to load the constant from the memory.

notation	operation	meaning	comments
add Rd, Rs, Rt	Rd $\leftarrow$ Rs + Rt	add	arithmetic op.
addi Rt, Rs, Imm	Rt $\leftarrow$ Rs + Imm[a]	add immediate	arithmetic op.
addu Rd, Rs, Rt	Ri $\leftarrow$ Rs + Rt	add unsigned[b]	arithmetic op.
mult Rs, Rt	[hi, lo] $\leftarrow$ Rs $*$ Rt	multiply[c]	arithmetic op.
and Rd, Rs, Rt	Rd $\leftarrow$ Rs AND Rt	bitwise and	logic op.
or Rd, Rs, Rt	Rd $\leftarrow$ Rs OR Rt	bitwise or	logic op.
sll Rd, Rt, shamt	Rd $\leftarrow$ Rt $<<$ shamt	shift[d] left logical	logic op.
lw Rt, offset(Rs)	Rt $\leftarrow$ MEM[Rs+offset]	load word	data transfer
sw Rt, offset(Rs)	MEM[Rs+offset]$\leftarrow$ Rt	store word	data transfer
beq Rs, Rt, Label[1]	if true: PC$\leftarrow$new	branch if equal	conditional
j Label	PC$\leftarrow$new	jump	unconditional

[a] Immediate (constant) [b] see tutorial notes [d] by shamt = shift amount

[c] multiplication will result in a longer product, which will be stored in dedicated registers, such as hi and lo

[6] writing to this register is not possible

[7] in reality, the programmer must consult the ISA to understand which registers can be used for temporary variables such as Rd

[8] recall the DFD examples with MUXes: any change in behaviour required a MUX

In table above we list examples of arithmetic and logic operations, data transfer (MEM means memory, the insn performs memory access) and operations that change the flow of the program (conditional branches and jumps). We will show some examples to clarify these instructions further.

1.4 Detailed discussion and MIPS 32 case-study

Instruction set design.[9] The instruction set design specifies many of the functions performed by the processor. Through the instruction set, the programmer (in case we refer to a lower abstraction level such as assembly programmer) can control the processor. The set of instructions define exactly what are the sequence of operations executed by the processor. All these definitions directly impact on computation performance. The definition of an instruction set involves:

- Instruction format, which include the instruction length in bits, field sizes and position of each field in the instruction. For example, MIPS has only three types of instructions (R-type, I-type and J-type) while RISC-V has five types of instructions (R-type, I-type and J-type, S-type and B-type).
- Registers: the number of registers in the processor is also defined by the ISA. In the instruction bits, each register is assigned by a specific code.
- Addressing: this defines the number of possible address fields and number of operands in an instruction.
- Number of arithmetic and logic operations, and their complexities.
- Data types to perform the operations.

Machine instruction fields. Each instructions type contain specific binary fields with fixed lengths. Usually, they are:

- **Operation code or function**: this is usually referred to as opcode or mnemonics (for assembly). For some instruction, this field is divided into operation code and function (see R type instruction from MIPS ISA for instance). The opcode usually defines the type of instruction and the function defines what the instruction does (i.e., addition, multiplication, etc.).
- **Source reference operand**: this field indicates the location of an operand in the memory. The memory can be be a register or a main memory address. This field can appear twice in the instruction (i.e., R type instruction in MIPS ISA).
- **Result operand**: the instruction produces a result that is placed in the field indicated by the operand results. This field appear once in the instruction fields.
- **Next instruction address**: for some of the instruction types (i.e., J type instruction in MIPS ISA), this field indicate the memory address of the next instruction to be fetched by the processor.

Types of operands. After the processor interprets an instruction, it will identify the fields indicating source and destination operands. As previously mentioned, the source and destination operands contains the label of a register where the actual operand is located. After the processor knows where the operands are located (i.e., what registers), it can perform arithmetic and logical operations over them. Usually, modern processors are able to process different types of data, such as *address*, *numbers*, *characters* and *logical data*. In the end, everything is binary data written in memory. For a processor, it does not matter if some data fetched from memory is a number of a character. The type of data is only important for the instructions.

Operands. Source and destination operands may be taken by a processor from different places:

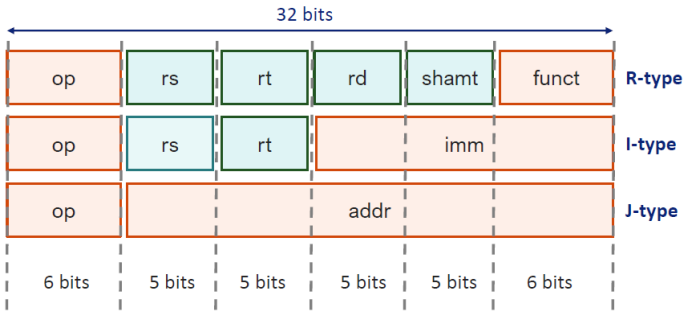
- **Main or virtual memory:** when operands are stored in main or virtual memory[10], an address is required to determine the precise location from which to read the operand (if it is a source operand) or where to write the operand (if it is a destination operand).
- **Processor register:** while a program executes, intermediate values are held in the processor's registers. To access the operand in a processor register, the specific reference (such as code or name) of the register must be provided for reading or writing operations.
- **Immediate**: in this scenario, the operand's value is directly written into the instruction's field. It is immediately available for processing without the need for memory or register access.
- **I/O device**: operands can be read from or written to the memory of an I/O device. The addressing rules for this scenario are like those used for main or virtual memory.

MIPS (Microprocessor without Interlocked Pipelined Stages). MIPS, which stands for Microprocessor without Interlocked Pipeline Stages, is a type of microprocessor architecture that has played a significant role in the evolution of computer systems. Developed in the 1980s by MIPS Computer Systems, the MIPS architecture is known for its simplicity, efficiency, and widespread use in various computing devices. In the next subsections, we will consider MIPS as an study case for a real Instruction Set Architecture (ISA). Modern ISAs, such as RISC-V or ARM would be more aligned to real-world and modern applications. However, MIPS, due to its simplicity, becomes an interesting case for learning purposes and for the understanding of how real ISAs can be used to implement any kind of program. Here, we will focus on the MIPS type that defines a 32-bit architecture. This means that all registers and instructions are 32-bit long.

[9] Adapted from notes by Guilherme Perin

[10] we will discuss wirtual memory on the transition into Topic 3

MIPS instruction set follows the principle that "Simplicity favors regularity", i.e., it keeps all instructions with the fixed bit length in order to simplify the design of a processor architecture. An instruction set that would consider instructions with variable lengths would be possible, however this would have a significant impact on the architecture's complexity. MIPS defines three types of instructions: R-type, I-type and J-type. Figure below illustrate the instruction fields: as we can see, the three instructions types have 32 bits and they have different fields.



R-type Instructions. R-type instructions form a crucial category that includes a variety of arithmetic and logical operations. The R-type instruction contains the following fields:

- op (6 bits): the opcode, which is always 000000 for the R-type instruction.
- rs (5 bits): the source operand. Its value correspond to the register in the RegFile.
- rt (5 bits): the source operand. Its value correspond to the register in the RegFile.
- rd (5 bits): the destination operand. Its value correspond to the register in the RegFile.
- shamt (5 bits): this field indicates the "shift amount". It is useful when the instruction requires logical shifts or rotations. The shift is applied to the source operands.
- function (6 bits): this field defines the type of arithmetic or logic operation that is implemented by the instruction. For example, function = 100000 indicates that the instruction does addition.

I-type Instructions. The I-type instruction are instructions with *immediate* actions. It contains the following fields:

- op (6 bits): the opcode, which defines the type of arithmetic or logic operation of the instruction.
- rs (5 bits): the source operand. Its value correspond to the register in the RegFile.
- rt (5 bits): the source or destination operand. Its value correspond to the register in the RegFile. For some I-type instructions, *rt* is the source operand, while for other instructions, *rt* is the destination operand.
- imm (16 bits): the immediate operand. The immediate operand can also be used as source operand by the CPU. The advantage here is that this instruction fields does not necessarily need to contain the location of the operand (the register name or an address value), but the operand value itself. For some instruction, the immediate field indicates the offset of a memory address.

J-type Instructions. The J-type instruction allows for unconditional jumps, and it is often used for implementing control structures like loops and function calls. It contains the following fields:

- op (6 bits): the opcode, to distinguish between different J-type instructions.
- addr (26 bits): target memory address.

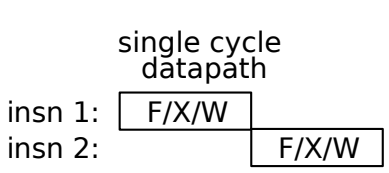
2 The CPU datapath

2.1 First Optimizations and Performance Analysis

In this section we begin the discussion on the CPU design. From the high level perspective, the CPU performs three steps: fetch the insn (F), execute the insn (X), and write back the results (W). Linking to Topic 1, we now consider insn to be the parcel, hence our throughput calculation becomes $\mathrm{Tput} = \frac{\#parcels}{\#cycles} = \frac{insn}{cycle} = \mathrm{IPC}$, which stands for instructions-per-cycle. Another important metric is the *cycles-per-instruction* CPI, the inverse of the IPC. Both values are related to the microarchitecture: we will show examples where different insns have different CPIs and how that affects the performance of the CPU. We will also consider the clock period.

It is natural that the three steps (F, X, and W) require different hardware, for example fetch will need to implement memory access and execute an ALU. The most natural option is to implement the CPU datapath as a single large combinational circuitry[11]. We call this approach *single cycle datapath*: it executes the three steps atomically[12] in a single cycle (F/X/W), then proceeds to the next insn[13]. This case is shown in on top of the figure below. While it has CPI and Tput of 1, the two values are a bit misleading, as the datapath had a long critical path and thus a large clock period. However, for very simple CPUs, single-cycle datapath implementations can still be found in practice.

The large clock period is a limitation. As an optimization, we can add a few registers to split the computation across multiple cycles (*multicycle datapath*[14]). Steps F, X, and W still reuse a lot of combinational circuitry and the registers are added to hold intermediate results, for example, step X will store the result of the ALU operation into a register before proceeding with W step. The registers are placed strategically to balance the critical paths throughout the CPU and minimize the clock period: a good trade-off is important because cutting the datapath into very short cycles also complicates the design of the control unit and may add too many registers[15], both of which will increase the CPU area. Note that while we obtain a small clock period, we "pay" the cost with a small throughput and bigger CPI.

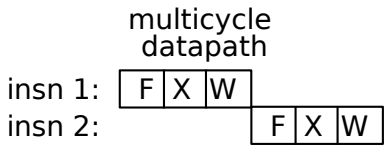


- true atomic F/X/W loop

  CPI $= 1$ (cycles-per-instruction)

  $Tput = 1$ (instructions-per-cycle)

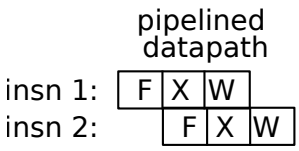
  t_{clk} = large



- one insn over multiple cycles

  CPI > 1 (and different for different insns)

  $Tput$ = small and t_{clk} = small



- overlap as much as possible

  CPI $= 1$ and $Tput = 1$ and t_{clk} = small

  NOTE: actual CPI and Tput are worse due to stalls

Finally, to overcome the limitations of small throughput and bigger CPI, we can pipeline the datapath to overlap the execution of the insns as much as possible. This allows us to keep a small clock period, while increasing both the throughput and the CPI. Again, this design option requires modifications to both, the datapath and the control unit of the CPU. Note that while the ideal Tput and CPI both equal 1, in reality they are lower: this discrepancy is caused by pipleine stalls, which will be discussed together with data dependencies. Recap from Topic 1: we will only consider fully pipelined datapaths: there is no reuse, and all stages must contain separate hardware[16]

[11] with exception of RegFile and registers used in the control unit

[12] without further separation

[13] insns are executed in an atomic F/X/W loop

[14] although we have three steps F,X,W, this does not imply 3 cycles

[15] both datapath and the control unit contain registers

[16] In Topic 1 we encountered replicated building blocks of the same type in multiple stages, such as adders. We think of the fully pipleined CPU datapath as containing replicated building blocks of different types in multiple stages, foe example, RegFile in one stage and ALU in another.

Metrics used for comparisons. A program being executed on a CPU is a a sequence of insns. We use Insn Count (IC) to denote the # of insns in the program. The execution time[17] T of a program running on a given CPU is defined as

$$\mathrm{T} = \mathrm{IC} \times t_{\mathrm{clk}} \times \mathrm{CPI} = \frac{\#insn}{program} \times \frac{\#seconds}{cycle} \times \frac{\#cycles}{insn}$$

Notice that the first and last fraction together yield the latency in terms of $\frac{\#cycles}{program}$ and we obtain the the total execution time we have seen in Topic 1, that is $T = L \times t_{\mathrm{clk}}$. The performance[18] is defined as the inverse of the execution time $Perf = \frac{1}{T}$. We will use **speedup** to compare different design[19] options:

$$\mathrm{Speedup} = \frac{\mathrm{T_{old}}}{\mathrm{T_{new}}} = \frac{\mathrm{IC_{old}} \times t_{\mathrm{clk-old}} \times \mathrm{CPI_{old}}}{\mathrm{IC_{new}} \times t_{\mathrm{clk-new}} \times \mathrm{CPI_{new}}}$$

The following observations hold for this equation:

- The IC depends on the program being executed. In case of compiling a high-level programming language code into assembly, using different compiler flags or different compilers can change the IC for the program. Similarly, manual modification of the assembly code change the IC.
- The t_{clk} depends on the microarchitecture. Modification to the hardware can change the clock period (recall Topic 1).
- The CPI in the equation above is the *average CPI* and is computed as $\sum_i f_i \cdot \mathrm{CPI_i}$, where $\mathrm{CPI_i}$ is the CPI for instruction i and f_i is the frequency of this insn in the given program. Hence, the average CPI may depend on IC and the insn. mix in the program and on the microarchitecture, which defines the $\mathrm{CPI_i}$ for each insn. Recall that for multicyce datapaths, different insns can have different $\mathrm{CPI_i}$s.

Amdahl's law, plays a crucial role in assessing the potential achievable speedup of the overall system (total speedup S_{total}). It quantifies the design principle *make the common case fast*: the impact of any optimization of a part X of the system is closely related to the fraction of time this part is being used. Namely, the more commonly used, the bigger the impact on the overall system. Let f be the fraction of part X being used in original system and S the speedup/optimization of part X, then the speedup of the entire system is computed as

$$S_{\mathrm{total}} = \frac{1}{\frac{f}{S} + (1 - f)}$$

2.2 Single cycle CPU example

The figure below shows an abstracted[20] schematic of a single-cycle CPU. We will first list the major components in the CPU (from left to right):

- **instruction cache** (I$) is a small memory[21] containing the instructions. The contents of the PC register are address used for accessing I$, and the I$ will react by placing the insn found on that address on its output (insn fetch step).
- **the control unit** is shown in red on the bottom, and is even more abstract then the rest of the schematic. It accepts the insn from I$ and stored it in an internal register called Insn Register (IR) until the CPU completes it and the next insn can be fetched.
- **the RegFile** is a small one-dimensional array of registers that are 32-bits wide. It holds the operands and provides them as inputs to the ALU and accepts ALU results (data). This is one option for the write-back step.
- **the Arithmetic-Logic Unit** (ALU) performs the ALU operation encoded in the insn (insn execute step).
- **data cache** (D$) is a small memory[21] containing the data. The address for accessing D$ is always provided by the ALU, even if the address was not modified by the ALU (merely passing through). The data is provided by the RegFile. The D$ will react according to desired direction by:
  - (CPU reads data from MEM) placing the contents (data) found on the address on its output. The data is then written into the RegFile (write-back step).

[17] response time

[18] similarly, the frequency is defined as the inverse of the clock period, hence frequency is a performance metric

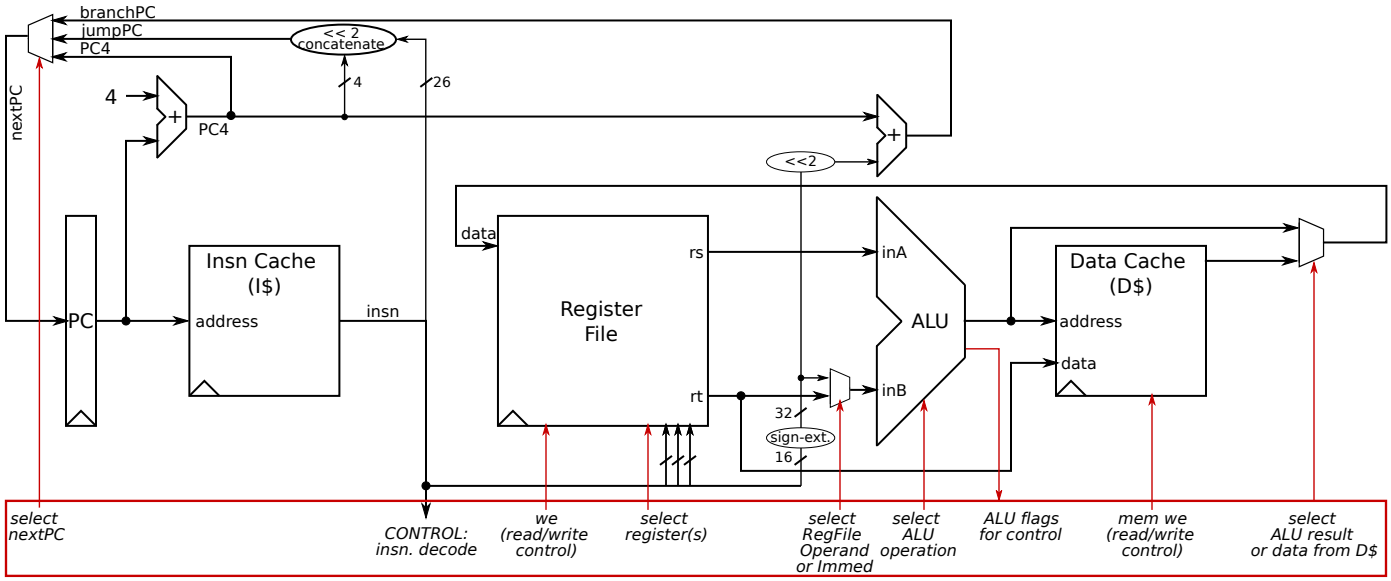
[19] we consider ISA design and microarchitecture design

[20] not showing all complex details, only the most important components

[21] we made a jump to the split memory, but for now let us just think of it as a part of the main memory with insns only, and we will explain details in the memory hierarchy chapter

– (CPU writes data to MEM) writing the contents on the data input of the D$ to the memory location indicated by the address on on the address input to the D$. For simplicity we will assume there is no output D$ in this case[22] and consider the write-back step as completed.

Next we will show some examples of insns and follow them through the CPU datapath. IF we do not mention particular parts of the datapath or particulat control signals, it means that for the given insn they are not being used[23].



2.2.1 Example 1: arithmetic

Let us look at the example `add Rd, Rs, Rt`, that is the addition operation Rd ← Rs + Rt, and assume the CPU already performed the fetch step (the insn is already in the control unit). The control unit will decode the insn and extract the values of Rs, Rt and Rd and set them as control signals into the RegFile (see "select register(s)"). The RegFile will put the contents of registers Rs and Rt on the outputs towards the ALU. The control unit will also set:

- the "select RegFile Operand or Immed" control signal to "RegFile Operand"
- the "select ALU operation" to addition[24]
- the "select ALU result or data from D$" control signal to "select ALU result"

The last control signal needed is the "we (read/write control)" for the RegFile, which has to be set to "write" in order for the RegFile to update the register indicated by Rd with the new sum. This value will be available for use in the next clock cycle.

2.2.2 Example 2: data transfer

We will talk about two cases: load word and store word. This naming convention is from the perspective of the CPU: the CPU will load the word into the RegFile (fetching the data into the CPU) or store the word into the memory (sending the data out of the CPU):

- lw (load word): will *write* into the RegFile but *read* from D$
- sw (store word): will *read* from the RegFile but *write* into D$

load word example: `lw Rt, offset(Rs)`, that is Rt ← MEM[Rs+offset], and assume the CPU already performed the fetch step. MEM refers to D$, that is the computed address will point at a location in D$. The control unit will decode the insn and extract the values Rt, Rs, and offset[25]. The RegFile will put the content of register Rs on the output towards the ALU. The control unit will also set:

[22] in reality this depends on the implementation of the D$, but the remainder of the CPU ignores possible output as the control signals prevent their usage

[23] they are treated as "don't care"

[24] see the R-type insn format: the operation is extracted from the "funct" field

[25] see the I-type insn format: the offset is contained in the "Immed" field

- the "select RegFile Operand or Immed" control signal to "Immed[26]"
- the "select ALU operation" to compute the correct address[27]
- the "mem we (read/write control)" for D$ to "read" in order for the D$ to place the conents from the memory word indicated by the address input on the output of D$, but at the same time to ignore the value on data input of the D$ (conetents of the D$ are unmodified by the lw insns).
- the "select ALU result or data from D$" control signal to "data from D$"

The last control signal needed is the "we (read/write control)" for the RegFile, which has to be set to "write" in order for the RegFile to save the new value into the register indicated by Rt. This value will be available for use in the next clock cycle.

store word example: `sw Rt, offset(Rs)`, that is MEM[Rs+offset]← Rt, and assume the CPU already performed the fetch step. The control unit will decode the insn and extract the values Rt, Rs, and offset. The RegFile will put the contents of registers Rs and Rt on the outputs towards the ALU, but the notice that the rt output splits into two directions: one for inB input to ALU and for the data input to D$ (the latter is important for the sw functionality). The control unit will also set:

- the "select RegFile Operand or Immed" control signal to "Immed[28]"
- the "select ALU operation" to compute the correct address
- the "mem we (read/write control)" for D$ to "write" in order for the D$ to save the new conents from the data input to the memory location indicated by the address input sf D$.

The last control signal needed is the "we (read/write control)" for the RegFile, which has to be set to "read" in order to prevent the RegFile from saving the value on data input of RegFile (conetents of the RegFile are unmodified by the sw insns).

2.2.3 Example 3: control flow

The program being executed on the CPU is a sequence of insns and we know that the CPU holds the address of the insn in the special register PC. This address a_i is used to access the I$ and retrieve the insn i. While the insn i is being executed, the address of the next insn $i+1$ is computed. Normally, this would be $a_i + 1$. In reality $+4$ as we are working wit 32-bit architecture, which means that all insns are 32 bits long and since addressed are encoded in bytes, that is 4 bytes. In the schematic, this address update is implemented with a small dedicated adder and contact 4, and it produces the value denoted PC4, namely $PC4 \leftarrow PC + 4$. There are other possible new addresses and top left corner of the schematic shows a MUX chooseing between three values for the *nextPC*; the control unit sets the "select nextPC" control signal based on the current insn being processed.

An example of an insn that modifes the control flow is the `beq Rs, Rt, Label` insn, that if Rs and Rt values are equal will "branch" to the new insn at address *new*. The value *new* is computed as follows: the Label is encoded as 16 bits as a part of the insn[29] and must be extended to 32 bits before being used (see seign-extend in the schematic between the RegFile and the ALU). It is then multipied by 4 (see $<< 2$ in the schematic), and used as one of the inputs to a small deicidated adder shown above the ALU. The second input is the value $PC4$, and the resulting 32 bit address is called *branchPC* in the schamtic. The control unit will decode the insn and extract the values of Rs and Rt and set them as control signals into the RegFile (see "select register(s)"). The RegFile will put the contents of registers Rs and Rt on the outputs towards the ALU. The control unit will also set:

- the "select RegFile Operand or Immed" control signal to "RegFile Operand"
- the "select ALU operation" to perform the comparison, and the output of the comparison is reflected in the dedicated output (ALU flags). The ALU will set a flag and the control unit will react accordingly[30] by either setting the "select nextPC" MUX control signal to *branchPC* or not.

A detailed comment: MIPS32 has only the "zero" flag, but other ISAs may specify other flags. Among the most common ones are "sign" (sometimes also called "negative"), "carry" and "overflow". While the ALU circuit details are not implortant for this course, a way to evaluate `bne` when only "zero" flag is available is by subtracting the two values being compared.

[26] the 16-bit offset needs to be extended to 32 bits before entering ALU
[27] see the I-type insn format: the operation is extracted from the opcode field
[28] the value from the RegFile is ignored
[29] see the I-type insn format
[30] based on flag value and insn

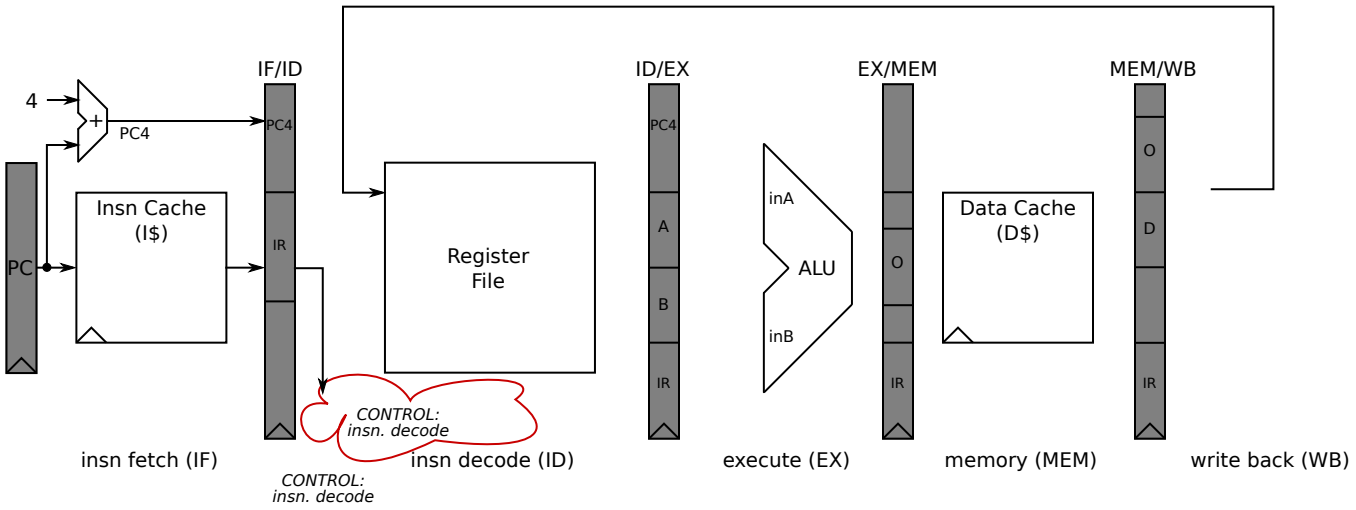
An example of an insn that modifes the control flow unconditionally is the jump insn `j Label`, that will always cause a jump to the new insn at address *new*. The value *new* is computed as follows: the Label is encoded as 26 bits as a part of the insn[31] and must be first multipied by 4 (see $<< 2$ in the schematic) and then concatenated with the 4 MSB bits of the $PC4$ value, resulting in 32 bit address called *jumpPC* in the schamtic. The control unit will decode the insn and set "select nextPC" MUX control signal to *jumpPC*.

2.3 Pipelined CPU example

We now begin discussion about a pipelined CPU. The figure below is an abstracted copy of the single-cycle CPU datapath discussed in the porevious section. As mentioned before, in order to increase the Tput and decrease the clock period, we can pipeline the datapath. Below we show the 5-stage pipeline with only the most important components of each stage. We omit a lot of details, such as wires and control signals. The five stages are named after their main functionality (which also implies the major components in the stage):

- Insn Fetch (IF): I$
- Insn Decode (ID): decoder[32] and RegFile
- Execute (EX): ALU
- Memory (MEM): D$
- Write Back (WB): logic to select new data for the RegFile

The interstage boundaries are shaded grey and named after the two stages they separate (with the exception of the PC register). Recall that we consider the insn as the parcel. As the insn moves through the pipeline so does its accompanying information, such as the incrememnted PC (registers PC4 in interstage bounaries IF/ID and ID/EX) and the insn itself (registers IR in interstage bounaries IF/ID, ID/EX, EX/MEM, and MEM/WB). Also note that since there is now an interstage boundary between the RegFile and the ALU, the two data outputs of the RegFile must be registered as well (registers A and B in ID/EX). Similarly holds for the ALU output O and D$ output D. Note that the path from WB back to the RegFile bypasses interstage registers. As an exercise, revisit the insn examples from the previous section and track their execution through the pipelined datapath.



2.3.1 Timing diagrams

We will use timing diagrams to show how the parcels (insns) move through the pipleine stages. A small example is shown on the right.

cycle	0	1	2	3	4	5
add Rd, Rs, Rt	IF	ID	EX	MEM	WB	

[31] see the J-type insn format
[32] a part of the control unit

NOTE: different notation w.r.t. Topic 1!!! As a consequence, we can not in the minimal example on the right, the cells after WB are empty! When we consider thext insn, that is insn 2, we must leave empty cells on the left. Namely, insn 2 can enter IF stage in cycle 1, that is after insn 1 completed the IF stage (cycle 0).

It is very important to understand that all insns must go through all stages! For example, the add insn does not need D$, but if add would skip MEM stage we would obtain a structural hazard with two insns trying to use the same hardware resource (stage) in the same clock cycle (cycle 4), as show in example below:

cycle	0	1	2	3	4	5
lw R7, 0(R6)	IF	ID	EX	MEM	WB	
add R3, R1, R2		IF	ID	EX	WB	

2.4 Data Dependencies

We have briefly mentioned data dependencies during Topic 1, however, we were reffering to dependencies between the values belonging to the same parcel. In this section, we will think about data dependencies between parcels (insns), more specififcally, between their operands and the results. The insns in question will be reading/writing the **same** register. We will show them on simple examples with two seonsecutive additions, more complex examples will be solved in tutorials.

- Read-after-Write (RAW)

  R3 $\leftarrow$ R1 + R2

  R7 $\leftarrow$ R1 + R3

  The result if insn 1 (write R3) is used as the operand in insn 2 (read R3). The correct order w.r.t. the value for register R3 is read-after-write, meaning that insn 2 must wait for insn 1 to complete.

- Write-after-Read (WAR)

  R3 $\leftarrow$ R1 + R2

  R1 $\leftarrow$ R7 + R5

  One of the operands from insn 1 (read R1) is over-written with a new value by the insn 2 (write R1). The insn 1 must read the *old value* from R1, and this dependency will not cause any problems as long as the two insns are executed in the correct order.

- Write-after-Write (WAW)

  R3 $\leftarrow$ R1 + R2

  R3 $\leftarrow$ R7 + R5

  Both insns are writing the same register R3. Insn 2 must overwrite the result written by the insn 1, that is, insn 2 must write *after* insn 1 is completed. This dependency will not cause any problems as long as the two insns are executed in the correct order.

We skip Read-after-Read as none of the two insns tries to modify the data, hence no dependency exists.

2.4.1 Pipeline stalling

In the simple example with two insns below we see the RAW dependency on R3: insn 2 reads the result of insn 1 as an operand. This dependecy is also called *register data hazard*[33], and must be resolved to ensure correct execution of the program.
Question: are *memory data hazards* possible in this pipeline ? No, because only MEM stage[34] is allowed to access D$.

From the CPU schematic we know that insn 1 must complete its WB stage for the correct sum to be available in R3. Insn 2 will complete IF stage, but will not be able to proceed through ID because of this dependency. Instead, the control unit will *stall*[35] insn 2 and allow insn 1 to continue:

[33] dependency that causes a problem is a hazard
[34] and insn currently in MEM
[35] terms stall, NOP and bubble are used interchangeably

cycle	0	1	2	3	4	5	6	7	8	9
add R3, R1, R1	IF	ID	EX	MEM	WB					
add R7, R1, R3		IF	◯	◯	◯	ID	EX	MEM	WB	

The control unit detects the dependencies by comparing Rd bits from the IR registers between different stages, that is IR values belonging to different insns, and takes appropriate action. It stalls the insn in question untill all dependencies are resolved. Extra hardware is needed for detection and stalling: the control unit starts sending NOPs (no operation), down the pipeline from the stalled insn onwards. NOP will not "change" anything, for example it can not write into D$ or RegFile.

2.4.2 Pipeline and bypassing

If we track the example above through the pipeline schematic we will notice that the correct result already exists in the pipeline as early as in clock cycle 3. It is stored in the O register of the interstage registers EX/MEM. Instead of stalling we can *bypass* the value to the input of the ALU:

cycle	0	1	2	3	4	5	6	7	8	9
add R3, R1, R1	IF	ID	EX	MEM	WB					
add R7, R1, R3		IF	ID	EX	MEM	WB				

We allow the following bypassing:

- EX/MEM to EX (both ALU inputs)
- MEM/WB to EX (both ALU inputs)
- MEM/WB to MEM (D$ data input)

The next schematic shows the bypass logic from MEM/WB to MEM (D$ data input) added in green: a new control signal is needed for the MUX selecting between the "normal" pipelined input from the RegFile (passed through register B) and the value bypassed from MEM/WB to MEM (D$ data input). Also note that the "normal" control signals are being passed on together with the insn through the interstage registers (ctl), but the bypassing control signals are not pipelined.