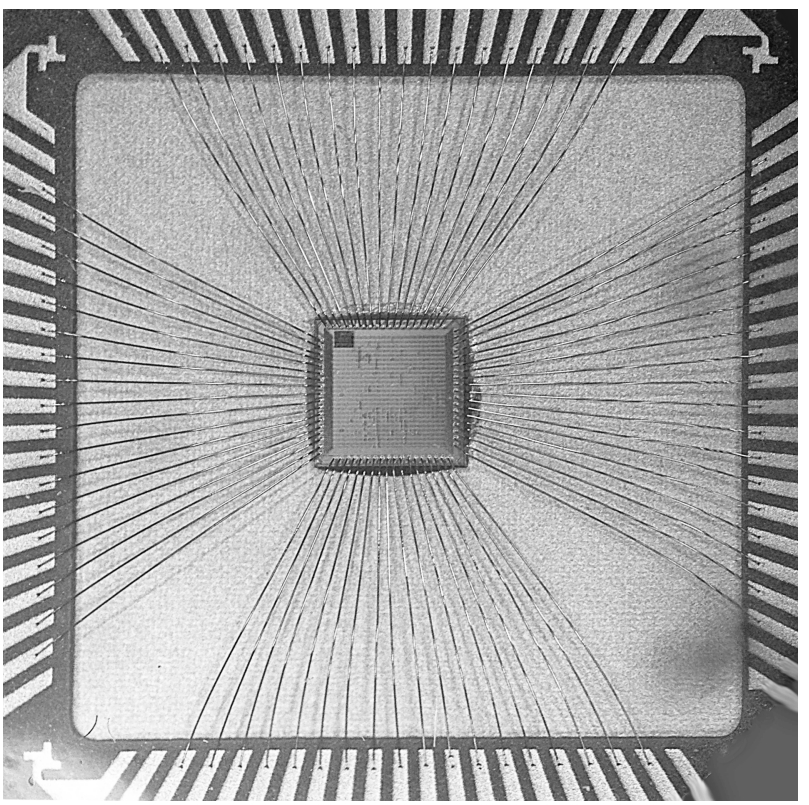
The Heart of The Machine 3



65nm CMOS ASIC, NOMAD Microprocessor, mounted in glass-covered test package. Photo: Christopher Crispin-Bailey 2019.

3.1 Processors: past, present, and future . . . 20
3.2 So what does a CPU do? 21
Heat and power . . . 22
What exactly is computation? 24
Toward a better definition 25
3.3 The performance barrier 26
Processor frequency 33
The power of multiples 34
Guesswork: prediction and speculation. 35
Branch prediction . 36
Speculative execution38
3.4 CPU internal architecture 38
An instruction 'in flight' 41
A more advanced microarchitecture 43
3.5 A little bit of programming 44
3.6 Extending the Instruction Set 47
3.7 More on instruction microsequences 48
3.8 Summary 49
3.9 Terminology introduced in this chapter 50

3.1 Processors: past, present, and future

In this chapter we will review the origins of electronic computer systems, and discover the key concepts behind the workings of the processor: the heart of the machine.

Since the mid 1970's, the vast majority of computer systems have been built around a key component: the single-chip **microprocessor**. Prior to this time, computer systems were almost always built from a large number of **discrete components**. These were often individual transistors in small metal canisters, or very simple digital logic components implemented individually as very simple logic chips[13] also known as **integrated circuits**. But even further back, computer circuits were built from **radio valves** filling hundreds of racks, and a multitude of cabinets.

[13] The most successful of these was the TTL logic chip family, which is still available and still used in digital electronics.

In terms of dimensions, a radio valve[14] is many times larger than a transistor, and a transistor is many times larger than a logic IC. A microprocessor is many many times more complex than a logic IC, yet of similar size. This is illustrated nicely in Figure 3.1 where some of these components are shown together at the same scale.

[14] As its name suggests, a radio valve was a component originally designed for driving radio amplifiers, but was also found useful in a variety of electronics applications, including implementing simple boolean (binary) logical functions.

Consequently, the march of progress in computer systems has been characterised by unrelenting miniaturisation: from valves to transistors, from transistors to IC's, and finally from IC's to microprocessors. Throughout this evolution of these important fundamental building blocks, the concept of a central processing unit (CPU) has been a constant feature.

A CPU in a 1950's computer might have consisted of many yards or metres of shelves full of radio valve racks, all wired together into a complex circuit, consuming kilowatts of power and generating more heat than an electric fire. In contrast, a modern day CPU is represented by a single microprocessor IC, may consume only watts or tens of watts, and yet such a chip is unimaginably more complex.

Whilst early CPU's might contain several thousand radio valves (in some way analogous to a few thousand transistors), a modern microprocessor may contain several thousand million transistors, and yet represent much more than just a primitive central processing unit. It is even possible for a complete computer system to fit on a single chip, with little else needed but a keyboard and screen.

Interestingly, heat and power are still a problem for modern CPU's, and this has a major impact on the design of modern computer systems and

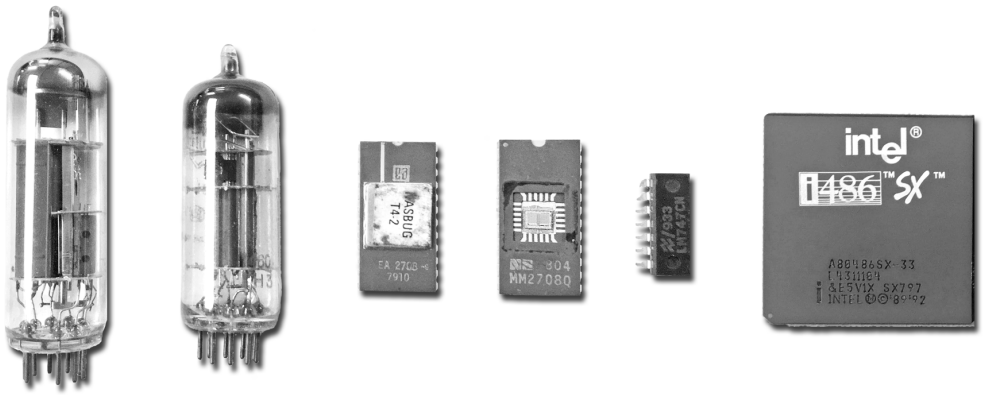


Figure 3.1: Electronic components, ancient and modern. Showing (left) radio valves roughly equivalent to one or two transistors, (middle) integrated logic chips with potentially of the order of 1 million transistors, (middle right) small IC chip package typical of simple logic gate chips containing typically 100's of transistors, and (far right) one of the first widely used one-million transistor microprocessors (1998).

their components, whilst influencing many of the trends that processor design is now following.

3.2 So what does a CPU do?

One of the most fundamental questions in computer architecture is: how do we compute information? The simplest answer we can give is: **the CPU performs computations**. But how does this happen?, why does it consume power, why does it take time? These are just a few of the questions we would like to know more about. So let us begin with a conceptual description of a CPU:

Definition 3.2.1 ***CPU: Simplified Definition***
*A **CPU** is a digital electronic circuit, usually built from thousands or millions of transistors. It can hold a small amount of temporary data and perform mathematical operations upon them. A CPU can also sequence operations according to a program it reads from memory, step by step.*

So, a CPU performs a sequence of (often mathematical) operations upon temporary data it holds, and this sequence is dictated by a program found in memory.

We can expand upon this description quite considerably, and we will do so later, but let us start with this simple case and understand what it means for computer design and operation. In particular we will examine design goals and design problems, and see how these are addressed in modern computer designs.

3.2.1 Heat and power

It is important to appreciate that all digital circuits consume electrical power. Just as a light-bulb requires power to generate light, a digital circuit requires power to perform its function. The amount of power consumed depends upon several factors, not least of which is the kind of **silicon technology** the chip is manufactured with.

Another factor is how fast the circuit operates. Just like a car engine, the faster it runs, the more energy it consumes. Every time a transistor switches on or off, it consumes some power, known as **dynamic power**. The faster the on-off transitions occur, the more power is consumed. And what happens to all of this power? Classical physics tells us that energy cannot be destroyed, only transferred, therefore when a circuit consumes energy, one of the unwanted by-products is heat[15]. Incidentally, silicon chips also consume power when the transistors are idle, like a car motor ticking over whilst waiting at a road junction. This is known as **static power**.

[15] There are however other by-products. For example circuits generate radio noise as well as heat. This can be undesirable, and theoretically it can even been exploited as a security risk.

When digital circuits are manufactured, they make use of miniaturisation of transistors to pack many components into a tiny chip, often only a few millimetres in size.

[16] Moore's Law: For a more detailed discussion try Wikipedia 'Moore's Law'.

As **Moore's Law**[16] has predicted successfully for many decades, the number of transistors on a given chip area doubles every two years. This can only be achieved by halving the dimensions of transistors at each step, and therefore reducing chip area per transistor by four. Only 20 years ago the critical dimension for transistor design was still as large as a 1 or 2 microns (1-2um), whereas these days the figure can be as small as 0.01 microns, or 10 nanometres (10nm)[17]. This dimension is known as the **feature size** and we refer to each incremental advance in silicon technology as a **technology node**.

[17] A micron is one millionth of a metre, or one thousandth of a millimetre

Observing the state of processor technology (in 2019), it can be seen that processors are manufactured at a range of technology nodes, including 90nm, 65nm, 45nm, 22nm, being quite frequently used across the semiconductor industry at the time of writing.

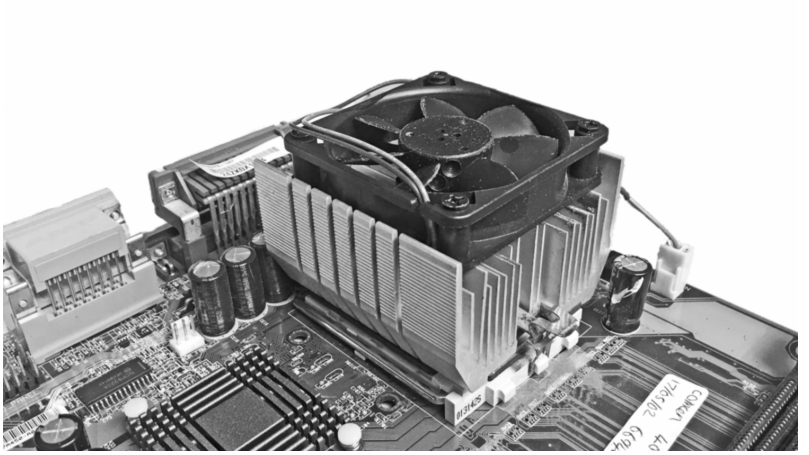


Figure 3.2: A processor with heat sink. Showing main CPU top-centre, with large aluminium multi-finned heatsink and forced air cooling fan. Lower left also shows smaller passive heatsink providing thermal dissipation for an auxiliary controller chip.

The power consumed by a transistor at each technology node reduces with size, so we might imagine that power is a diminishing problem for a digital circuit such as a processor. However, that is not the case. Why is this? The answer is that although transistors get smaller and smaller, designers want to use more and more transistors per chip, and those transistors operate faster too. The complexity of processors is thus an ever increasing feature of computing.

Another consequence of this trend is that heat generated by circuits increases too. Where we have smaller circuits consuming more power and thus generating more heat we can say that **power density** and then **thermal density** increases. This is recognised as a serious problem. This is why a modern processor system is often cooled by a large **heat-sink**, a module designed to draw heat away from the processor so that it does not overheat. An example of a heatsink is shown in Figure 3.2, where the example CPU requires a substantial **metal-finned heatsink** supplemented with a **cooling fan**. Some heatsinks are further assisted by liquid cooling systems.

Let us consider an example to put this in perspective: An INTEL Core i7 8700 3.2GHz chip is designed to run at up to 67 Watts with the recommended heatsink. So 15 of these processors could consume as much power as a 1kilowatt electric fire, and generate a substantial proportion of the same heat. That is a lot of power to pay for and a lot of heat to deal with.

The problem of heat also dictates some considerable design limitations. **Server farms**, huge warehouses full of racks of computers, produce so

much heat that they must be cooled by advanced cooling systems, and these in turn consume power. Therefore, the total power consumed, and thus the running cost of the servers may be much higher than just the CPU power consumption itself.

Other systems are physically constrained within modules: a laptop case, a system rack, inside a handheld phone or device. These systems cannot generate more than a certain amount of heat because they are not able to get rid of it faster than a certain rate. You may have noticed your smartphone getting warm when heavily used. This might well be the limiting factor in performance, and not how powerful the CPU itself is, or the size of the battery.

3.2.2 What exactly is computation?

Now that we have established that digital circuits are what makes a CPU work, but at the cost of heat and power, what exactly are these resource-hungry computations doing?

A typical CPU will perform a range of different computational operations, also known as **instructions**. These operations act upon data in different ways, such that if multiple instructions are combined into a **program sequence**, almost any computational task can be performed. Let us survey some common instruction characteristics:

Some characteristics of instructions

- Data is held in temporary storage components within the CPU, called registers. Instructions operate upon these register contents.
- Many operations are directly mathematical: add, subtract, multiply, divide, and so on.
- Some of these instructions operate only on integers (whole numbers) whilst other operations may be able to act upon floating point or fixed point numbers. Not all processors support both kinds of computation.
- Some instructions are logical operations. These perform boolean logic operations on data.
- Some instructions transfer data to and from memory. These are known as Load/Store operations.
- Some instructions test conditions and choose which part of the program to execute next.

We have just introduced some new concepts here: for example, what does **execute** mean? A program consists of a number of instructions, and a CPU **executes** (in other words, performs) each instruction in turn. In order to execute an instruction, the CPU must first **fetch** it from memory, where our program is usually stored. Fetching an instruction, and then executing it, is known as the **fetch-execute cycle**. The CPU repeats this over and over indefinitely (or until such time as it reaches a halt instruction to tell it to stop).

If a CPU simply executed one instruction after another in a continuous list, we would have what is known as a **linear program sequence**. However, very few programs can be written like this in their entirety. Actually most programs include choices and decisions which affect which part of the program is executed next.

Consider the comment that 'some instructions test conditions and choose what part of the program to execute next'. These are known as **program-flow instructions**. The flow of a fetch-execute cycle sequence, as it moves through memory, is generally **non-linear** (more accurately it is linear for short sequences), interspersed with a number of jumps to other parts of the memory where the next short linear sequence is located. Program-flow instructions are therefore frequently encountered.

3.2.3 Toward a better definition

We have gathered quite a lot of information about our initial question. So let us try to answer this again: What is a CPU?

Definition 3.2.2 ***CPU: Expanded Definition***
*A **CPU** is a digital electronic circuit, manufactured from a silicon technology defined as being at one particular technology node, consuming a certain amount of power, and generating a certain amount of heat during its operation. A CPU fetches program instructions from memory using a concept known as the fetch-execute cycle, and as each instruction is fetched, it is executed, performing one of a number of possible operations upon data held within the CPU registers. These operations include mathematical and logical operations, and operations that change the flow of the program to execute parts chosen by the result of testing conditions encountered during a program sequence.*

This is quite a detailed description, and hopefully one that makes sense given the preceding subsections. We should be able to appreciate now

that processors can operate at different speeds, and have varying degrees of power consumption, and they create heat, which is usually not desirable[18], but by slowing down processors, they can consume less power and create less heat at the cost of reduced performance.

[18] Heat is nearly always a nuisance. However, some researchers have used CPU's to heat water to power central heating systems using waste energy from computations.

3.3 The performance barrier

We will return to the idea of instructions shortly, and explore further. There are however some higher level concepts that we must understand a little better, not least because they have a major impact upon performance of modern computer systems.

What is performance?: performance is the achievement of certain goals whilst consuming certain resources. Another way to express this is cost versus benefit or **cost-benefit**. We sometimes refer to this as a **performance metric**. Let us consider some cost-benefit examples in very loose terms, making some assumptions:

- A given CPU can execute one million instructions per second (meaning it has a throughput of one **mips**)
- The CPU may well require 1 Watt/hour of power to do this.
- Assume that one watt of electricity costs 1/10th cent,
- Assume the CPU costs $100, and is used for 1 yr continuously.
- The CPU dimensions are 20x20x5mm

Now what can we say about this CPU?, it has a raw performance of 1 mips. This is a measure of how much work it can do in a given amount of time. But how about a cost-benefit analysis:

This CPU performs **1 mips per watt** (because we use 1W of power and get 1 mips in return).

Use of the CPU for one year requires the purchase price (100 dollars), plus the cost of 1 year of electricity at the stated rate of consumption. One year equates to 365x24 = 8760 hours, and therefore 8760 x 1W = 8.76 kWh. At 0.1 cent per watt-hour, this electricity will cost 8760 cents, or $87.60. Therefore, the total cost of owning and running this CPU for one year is approximately $188.

Why is this important? The reason we want to know about such things is because we want to be able to predict the **cost-effectiveness** of a given computer system, firstly as a singular case, but secondly in terms of comparison between two or more systems. Consider, for example, the list of performance characteristics given in Table 3.1.

	mips	W/hr	Dimensions	Cost of CPU
CPU A	10	40	20x20x5 mm	$280
CPU B	5	11	15x15x5 mm	$150
CPU C	5	15	25x25x6 mm	$80
CPU D	8	4	25x25x5 mm	$200

Table 3.1: Various imaginary processors and some of their attributes.

Given this data for multiple hypothetical processors, can we work out from this table, the following information?

- Which CPU can do the most processing in one hour?
- Which CPU is most cost efficient for power consumed?
- Which CPU gives the most performance per dollar?
- Which CPU is cheapest to buy and run for one year at 1 mips?
- Which CPU has the worst (highest) power density at peak mips?

Each of these is a performance measure often known as a **performance metric**. Try and answer these questions before reading on, and see how much you can answer correctly.

Most Processing Per Hour

We can say immediately that CPU A is the best choice for this objective. At 10 mips, it performs more work than any other processor in the list in a given period. We could calculate the total instructions executed per hour if we wish, but the same conclusion will be obtained.

Instructions executed in one hour

CPU A: 10 mips x 3600 seconds = 36,000 million instructions.
CPU B: 5 mips x 3600 seconds = 18,000 million instructions.
CPU C: 5 mips x 3600 seconds = 18,000 million instructions.
CPU D: 8 mips x 3600 seconds = 28,800 million instructions.

It is important to realise that mips can only measure raw performance. It does not automatically reflect the true performance of processors in general terms, since each process has a different instruction set.

Most Cost efficient for power

We can determine this using our mips-per-watt measure, or watts per mips. We can see that the most power efficient processor is the one with the lowest watts per mip, or the highest mips per watt, CPU D:

Most Power Efficient

given Mips per Watt = $\frac{mips}{watts}$ and Watts per Mip = $\frac{watts}{mips}$

CPU A: 10/40 = 0.25 mips per watt, and 4 watts per mips.
CPU B: 5/11 = 0.45 mips per watt, and 2.2 watts per mips.
CPU C: 5/15 = 0.33 mips per watt, and 3 watts per mips.
CPU D: 8/4 = 2.0 mips per watt, and 0.5 watts per mips.

We might wonder why we need two measures. Actually either of these metrics is sufficient to answer the question posed. However, if we want for instance to run a computer system on a battery with only 50 watts available before the battery charge runs out, then we can use watts per mips to quickly determine how much work any one of the processors can perform, or for how long a particular task can be performed (if we know the mips requirement).

Now, if we were to have a battery system that provides a maximum of 50 watts per charge, and we need 1 mips to run a particular task, CPU D can be seen to be able to run for 100 hours on one battery charge. [19]

[19] With a 50 W charge, 1 mips required to sustain the computational task, and 0.5 Watt per mips, therefore expends 50 Watts at 1 x 0.5 Watts per hour, thus 100 hours of operation is possible.

Most performance per dollar

This is another simple calculation, but having the same caveat as performance per hour. We are only measuring raw performance per dollar. CPU C has the best dollar per mips or mips per dollar:

Most performance per dollar

CPU A: 280/10 = $28 dollars per mips.
CPU B: 150/5 = $30 dollars per mips.
CPU C: 80/5 = $16 dollars per mips.
CPU D: 200/8 = $25 dollars per mips.

Least Expensive to buy and run for one year at 1 mips

For this calculation we must combine the *cost* of power consumed, with the cost of purchasing the processor. The least expensive processor to buy may not be the least expensive overall, conversely the one that consumes the least power per mips may not be the best either, depending upon the lifespan of the intended use. Indeed we see CPU C is the best choice here even though it is one of the poorer performing processors for power efficiency alone.

Most Cost effective to own and run

At 0.1 cents per watt-hour, a kWh[20] costs 1000 x 0.1 cents = $1.

CPU A: 4 watts per mips, therefore 365 x 24 x 4 watt-hours = 35.04kWh, costing $34.94.

CPU B: 2.2 watts per mips, therefore 365 x 24 x 2.2 watt-hours = 19.21kWh. costing $19.22.

CPU C: 3 watts per mips, therefore 365 x 24 x 3 watt-hours = 26.21kWh. costing $26.21.

CPU D: 0.5 watts per mips, therefore 365 x 24 x 0.5 watt-hours = 4.37kWh. costing $4.37.

And then, total costs (to nearest dollar):
CPU A: $280 + $34.94 = $315
CPU B: $150 + $19.22 = $169
CPU C: $80 + $26.21 = $106
CPU D: $200 + $4.37 = $204

[20] kWh : Kilowatt Hour, the typical measure for electricity cost calculations.

It is important to recognise, as with some earlier metrics, that there are additional factors not considered here. For example, a computer system contains a power supply module, cooling fans, and much more hardware. This calculation only considers the CPU in isolation. The benefit of choosing CPU C may thus be far less significant in overall terms.

Worst and best power density at peak mips For a complete processor chip in its packaging, power density is a function of volume and power consumption[21]. CPU C has the worst power density:

approximate power density

CPU A: Volume = 20x20x5 mm = 2000 cubic mm. at 40W this gives 0.02 W/mm^3 or 20 mW/mm^3.
CPU B: CPU A: Volume = 15x15x5 mm = 1125 cubic mm. at 11W this gives 9.8 mW/mm^3.
CPU C: Volume = 25x25x6 mm = 3750 cubic mm. at 11W this gives 4.0 mW/mm^3.
CPU D: Volume = 25x25x5 mm = 3125 cubic mm. at 11W this gives 1.3 mW/mm^3.

[21] There are other conventions for power density worth highlighting. In chip design, the actual chip die is effectively a 2-dimensional slab of silicon, of the order of 10s to 100s of square millimetres. In this domain power density is typically quoted in watts/cm^2 in relation to die surface area. Silicon engineers may even calculate power density in small regions of the chip where power can be very high (known as hot spots).

We can conclude that CPU A has the worst power density at peak mips and CPU D has the best. Power density can be important. Where power density is very high, special cooling requirements may arise such as heat-sinks and CPU fans, or even liquid cooling in some extreme cases[22]. In less demanding situations, devices that are hand held or wearable, and and therefore very compact, may also have difficulties even with more moderate power densities.

[22] Indeed, taking this to the full extent, server cabinets full of 'rack-mount' circuit boards can be rated in power capacities of the order of 20-30kW, and power densities calculated accordingly given the cabinet volume. This is used to indicate how much compute capability can be hosted in a given space. This is a critical concern in server farms and data-hosting centres, for example.

3.3.0.1 Power/Frequency Scaling

Given that power is such a significant concern, and indeed power consumed ultimately turns into heat creating a further problem, we would expect that CPU designers have spent much effort on trying to deal with these issues.

Unfortunately the ideal solution, of endlessly making transistors more and more efficient, thus consuming less power, is simply beyond the realms of physics as we understand them today. Circuits and chips *are* becoming more efficient, but nowhere near as rapidly as we would wish. Designers currently turn instead to optimisation strategies at another level in order to make the best of the systems as they are able to be fabricated.

Processors operate at particular clock frequencies, and each clock pulse equates to a cycle in which some work is performed, some power consumed, and some waste heat generated[23] . A simple solution to power consumption is to reduce clock frequencies when the CPU does not have much work to do, and to 'ramp up' the frequency when work suddenly becomes demanding. This may well be labelled with various conceptual terminology such as *speed-scaling*, *speed-step*, *CPU throttling*,and so-on. It might be considered analogous to turning a tap on more or less to moderate the flow of water.

[23] In actuality, not just heat. Processors also radiate radio frequency energy as part of the 'waste' energy output, though it is less often considered a problem.

If we take the frequency scaling idea to its extreme, then a CPU that typically runs at 100's of MHz might well only be clocking at 10's of KHz. In this mode the CPU is effectively in what is know as **sleep mode** and can only do as much as is required to keep an eye on what is happening in the system so it is ready to 'wake up' and ramp up the clock rate when needed. Sleep modes are often a little more sophisticated than this, turning off some parts of the CPU entirely for periods of time, might have multiple options, and can achieve very high power savings. However, as we always tend to discover in computer architecture, things are rarely as simple as turning a dial and getting a linear response. Remember, as

mentioned earlier, a CPU has two kinds of power consumption, **dynamic** and **static** power.

Dynamic power is the power that is consumed for each operation performed, such that the more operations you may execute per second, the more power is consumed. This is a more or less linear relationship for our purposes. Ten times as fast a clock rate means ten times the (average) dynamic power consumed[24] . On this basis, setting the clock rate to zero ought to reduce power consumption to zero too.

[24] Provided it is ten times the same kind of work. The precise amount of dynamic power consumed depends upon the exact mix of instructions being executed , with an average being an approximation.

Static power is power that is consumed continuously even when the CPU does nothing, even when its clock rate is set to zero (which many CPU's do not permit in any case). structures like register files, on-board memories, and other circuits, will consume power even when idle and not even clocked by a clock signal. This static power is like a baseline, upon which dynamic power is then added.

CPU complex power Example

Suppose a CPU has the following power data from the manufacturer's data sheet, and a maximum clock rate of 100MHz:

Standby (sleep) mode power: 0.1 Watt.
Static Power, P_S when active : 20 watt.
Dynamic power, P_D when active: 2 Watt per MHz (P_d=2W).

Assuming that Power = P_S+P_D , where $P_D = P_d$ x F MHz,

Then, we can see that this processor, when active, will consume up to 220 Watts at maximum clock rate of 100 MHz:

Total Power = 20W + (100 x 2W) = 220W

But we can also see that if we halve the clock rate, the power is then:

Total Power = 20W +(50 x 2W) = 120W

Notably, the total power has not halved, due to the residual static power that is always present. Importantly, total power is not linear with respect to frequency.

Now what about sleep mode? Suppose the CPU is clocked at full speed for 50% of the time and is in sleep mode otherwise:

At 100MHz, P_{tot} is 220W , but for 50% time , thus 110W.
And in standby mode: 0.1W x 50% = 0.05W.

Thus total power is 110.05W, better than simply scaling the clock.

What we have observed is that power can be moderated in different ways using CPU design features. In practice we have not considered all of the factors, there are drawbacks and penalties for moving in and out of standby mode or for changing clock frequencies 'on the fly'. Such factors vary between CPU designs, and these may well make marginal cases less worthwhile to consider. However, we do see that when a CPU has no useful work to do, or very little, it is potentially valuable to use standby modes.

3.3.0.2 More about Mips

A further word about **mips** at this point might be useful. What exactly do we mean by mips? In its simplest possible definition, mips is simply millions of instructions per second, a simplistic measure of the **computational throughput** a processor is capable of. However, in a more realistic situation, we would know what sort of instructions we are executing, how often, and so on.

Every program is different, and every processor has a slightly different instruction set, so a simple mips measure is not really a true representation. We may also come across the term **peak mips**, which measures the maximum possible mips a processor can achieve under the best possible conditions. Often when we talk of billions of instructions per second, it is common to use GOPS and GFLOPS (Giga-operations per second, and Giga floating-point operations per second), and these can also be peak measures in some cases.

To measure anything other than peak mips or peak gflops, and thus perhaps compare several processors with realistic code behaviours, we would need to have a common test program to run to reproduce the same workload on each processor in turn. These test programs are known as **benchmarks**.

A well known example, **Scimark** is a scientific computing benchmark which can be executed on any processor, and then used to measure identical work across many systems. There are different kinds of benchmarks for different kind of computing tasks, such as artificial intelligence, scientific computing, gaming and multimedia, embedded systems, and other domains. The lesson here is that a performance metric should be chosen carefully to test the behaviour that is relevant to the problem.

In a simplistic example, consider a program that multiplies two 3x3 matrices together. Suppose our earlier CPU A requires 1000 instructions to complete this task, and CPU B requires 800 instructions to complete the

	Frequency	CPI	Peak mips
CPU X	3 GHZ	1	3,000 mips
CPU Y	5 GhZ	2	2,500 mips
CPU Z	5 GHz	0.8	6,250 mips

Table 3.2: Hypothetical CPUs with clock rates in GHz, peak mips, and CPI.

same work (but using its own slightly different instruction set). We might think CPU B is best. But remember that CPU A performs at 10 mips (peak) and CPU B performs at 5 mips (peak). Will 1000 instructions at 10 mips complete more quickly than 800 instructions at 5 mips? Almost certainly yes in this case, but without running the two CPU programs (effectively a simple benchmark) we cannot be certain.

3.3.1 Processor frequency

Another key factor, relevant to performance measurement, is processor **clock frequency**, which is a measure of how fast a processor can perform a key internal circuit operation[25] . We measure frequency in **cycles per second**, or **Hertz**. Since most processors and digital circuits run at incredibly fast speeds, we typically refer to processors running in the megaHertz (MHz) or gigaHertz (GHz) range.

[25] This is usually the slowest circuit path within the design, known as the critical path.

This is one area where we must be particularly cautious: internal circuit operating frequency, or **core clock frequency**, is not directly proportional to mips. An instruction might require one clock cycle, or two, or three, or perhaps many more. The same identical operation on another processor might require more, less, or the same number of clock cycles.

Therefore, a higher clock frequency on its own is not a guarantor of higher mips or higher benchmark scores.[26] To unravel this potential confusion we need to know another important parameter: **clocks per instruction**, or **CPI**.

[26] An analogy is a car engine, where frequency is represented by revolutions per minute (rpm). Here rpm is not an indicator of speed on its own, we also need to know about the gearbox or transmission ratio, and even the diameter of the wheels, to determine how fast a given vehicle can move. So an identical rpm (frequency) could give easily very different results in two different vehicles.

Consider the data in Table 3.2. If we consider frequency alone, then CPU Y and CPU Z appear to have identical performance, and CPU X is not as good. But if we also consider the CPI rating of each processor, then we see that CPU Z is the best choice, and CPU Y is actually the worst choice for peak mips.

In order to make sense of this, we again face the problem of exactly which instructions we are referring to when we use CPI: Clocks per instruction?, yes, but clocks related to which instructions? Again, benchmarks help

us out here and a credible CPI measure would ideally be based upon a suitable benchmark executed on each CPU we wish to compare, using an appropriate mix of different instructions. This is in fact how many computer systems are rated for performance capabilities, and the benchmarks might be general purpose, scientific workload, multimedia focused, or based upon more specialised requirements.

3.3.2 The power of multiples

It may well appear that to some extent the obvious path for processors is to push for smaller and smaller transistors, more and more complex CPUs, which are faster and faster, leading to higher and higher clock frequencies, and greater and greater processing throughput, be it mips, scimark score, or any other performance metric. However, this trend, which has indeed been sustained for several decades prior to the 2010's, has a downside.

As transistors get smaller, their power consumption becomes more and more of a problem, design issues can cause CPI to increase, and the ability of more and more complexity to exploit the most frequently occurring cases in workloads becomes less and less beneficial[27] As a result of this, CPU's have shifted through several eras of new design thinking, often referred to as **paradigm shifts**. Whilst we do not need to have a deep understanding of these ideas here, it is very useful to understand the terminology and general principles, as these are ideas that come to the fore regularly in computer systems design and use.

[27] This is known as the Law of diminishing returns. Successive application of equal effort delivers progressively smaller benefits.

Early processors were able to execute one instruction at a time, and would be classed as having a **scalar execution model**. Here, the execution of instructions was serial, meaning that one instruction would complete before the next began. This seems fairly logical, if programs are sequences. However, it was recognised early on in the history of processor design that an instruction need not always be completed before the next one begins. This concept of **overlapped instruction execution**, a form of a technique known as **pipelining**, allowed more instructions to be executed within fewer clock cycles, meaning that the inherent CPI decreases: usually an indication of better performance capacity.

This kind of optimisation is only possible where the two instructions do not depend upon one another: the next instruction clearly cannot use the result of the previous instruction unless it waits for the preceding instruction to finish[28]. This potential dependency is known as a **pipeline hazard** or **data hazard**, and with care, and using a concept known

[28] When one instruction relies on the result of another, we call this a dependency. Only non-dependent instructions are potentially able to execute in parallel.

as **data-flow analysis**, code can be written to avoid many of these problems.

Once it was understood that instructions could overlap frequently enough for a benefit to be gained, the next progression was to realise that multiple instructions could start at the same time. This is referred to as **multiple issue**: issuing multiple instructions per clock cycle, and is the basis of the **superscalar execution model**.

In superscalar execution, instructions can be issued in groups, perhaps 2, 3, or 4 instructions at a time[29] and these instructions finish at different times (**out of order completion**). Of course, this means that instructions are issued in order but may complete out of order. Again, the consequences of instructions relying upon preceding results leads to data hazards of various kinds, but careful program construction or clever compilers will avoid these problems[30].

Eventually, superscalar processor issue width also hit a limiting point in its evolution. Moving from an issue width of 2 to 4 yielded good benefits but moving from 4 to 8 yielded a diminished return. The concept could not exploit the ever-increasing complexity available for chip designers, because there was not enough non-dependency in the instruction sequences. Therefore, another shift in architectural thinking was required.

The idea of placing multiple processors on one chip was the only obvious next step to use the larger transistor counts being offered. This resulted in the creation of the current generation of **multicore processors**, where perhaps between 4 to 8 large complex processor cores are integrated onto a single chip. As we will see later, this multiplicity of processor cores has relevance to modern operating systems, and introduces some additional advantages.

Hopefully what we can learn from this is that a superscalar processor might operate at a lower clock rate than a scalar one, yet execute more instructions, and a multicore processor could easily outperform a single core even when running much more slowly.

[29] This parameter is known as **issue width**. Sometimes processors are referred to as being 2-way or 4-way superscalar, which means they have an issue width of 2 or 4 instructions per clock cycle.

[30] There is a substantial branch of computer science devoted to instruction scheduling, and this is a topic with many avenues to read upon further. See also Tutorial **??**.

3.3.3 Guesswork: prediction and speculation.

As we have observed earlier, a program is typically non-linear in program flow. In other words, rather than instructions being executed entirely consecutively from memory locations 0,1,2,3,4,5 and so on, we find that a program sequence might be linear for a relatively short run, and then jump to another part of the program where a further linear sequence

of instructions is encountered[31]. Often these non-linear transitions, or **jumps**, are due to a **conditional evaluation** (in other words, a computation resulting in some form of comparison with a true/false outcome).

[31] This is often related to program structure in the high level language: such as loops and if-then-else statements.

For example if we tested if some data value 'X' is greater than zero, there are only two possible outcomes: true or false. A program may decide to jump to a new program section if the comparison is true or just continue linearly to the next instruction if the result is false. In this way, program sequences can become conditional upon the results of other program sequences.

As a result of this **conditional decision point**, the location of the next instruction to be executed is not always known in advance, meaning that we cannot fetch the next instruction whilst the current one is executing. Remember how we observed that early scalar processors progressed by introducing overlapped instruction execution: a simple form of pipelining, to achieve a performance gain? When linear code sequences are executed, overlapped instruction fetching is fine, but when a condition is encountered and a jump occurs, this presents us with a temporary roadblock, and time is wasted waiting for the comparison instruction to complete, costing the system some performance. This is known as a **branch penalty**.

3.3.4 Branch prediction

[32] In computer architecture a 'Branch' is just another name for a 'jump'.

We can deal with the problem of unknown jump conditions using a technique called **branch prediction**[32].This can be very simple or rather complex.

In the simplest case the CPU designer could simply assume that certain jumps are always true and others are always false. This fixed branch prediction scheme, known as **static branch prediction**, can be designed into the CPU circuitry or **hardwired**. It would rely upon the fact that, statistically, certain branches behave in certain ways. Of course this is primitive, and is only as good as the assumptions made in the first place.

The static branch prediction strategy may not work well for all programs. An alternative is to use **dynamic branch prediction**. In this scheme, the processor keeps track of how often particular branches are found to be true or false via a **branch history table**. Then, when the CPU encounters a branch, it uses the most up-to-date information in that table to decide if the branch should be taken or not. Call it an educated guess, based on the way the program is behaving at that point in time.

Clearly this scheme has some advantages: it would be expected to adapt to the behaviour of individual programs, and even particular parts of programs, and can therefore be more accurate than less sophisticated schemes[33].

[33] There are actually a whole variety of dynamic branch prediction algorithms, some better than others.

So what does branch prediction allow us to do? If we guess that a branch is taken or not taken, then we can begin fetching the next instruction immediately, with overlapped instruction fetching preserved, and performance kept at peak level. This may typically save one or two clock cycles. If the prediction is correct then we carry on and everything is wonderful! On the other hand, if the prediction is wrong, then that fetched instruction has to be discarded and the CPU then switches to the correct location (which by now it will know). This is known as a **branch mis-prediction penalty**, and this may be of the order of a few clock cycles typically.

What we are encountering here is a **performance tradeoff**. If a correct branch prediction saves one clock cycle and an incorrect prediction suffers a 2 clock cycle penalty, then how do we know we are getting a good deal? We can evaluate this if we know the branch prediction **hit rate**, which is the success rate of the prediction algorithm:

Hit rate impact example

Suppose for example we have the following information:

- The CPU predicts the branch correctly 80% of the time,
- For this particular CPU this saves 2 clock cycles,
- Meanwhile a mis-prediction also costs 2 clock cycles.

What is the net benefit?

Gains: 80% x 2 clock = + 1.6
Costs: 20% x 2 clocks = - 0.4
Combined: 1.6 - 0.4 = 1.2

Overall = +1.2 clocks (a saving of 1.2 clock cycles per branch).

So it is apparent that, in this example, we gain 1.2 clock cycles per jump/branch if the hit rate is 80%.

Now, can you determine what are the net gains or losses if the hit rate is 40%, 60% and 90%? Try and calculate these based on the example preceding*.

* Answers: 40% (-0.4), 60% (0.4), 90% (1.6).

We may well wonder at this point: is a saving of 1.2 clock cycles really such a big deal? It doesn't sound like much when we talk about processors running at 100's of millions of instructions per second. However, it turns out that branches are very common (as many as one in 6 instructions can be branches). So for a processor executing 1 million instructions per second, we might save:

$$1.2 \times 1{,}000{,}000 \ / \ 6 = 200{,}000 \text{ clock cycles per sec}$$

This is a lot of computation time and resource saved. You may wonder why branches are so frequent. This is because they are used heavily in program structures such as loops, which are a fundamental workhorse of programming practice.

3.3.5 Speculative execution

Speculative execution is an extension of branch prediction, particularly useful in superscalar processors. Here, rather than simply predicting the branch so we can fetch the next instruction, the CPU allows execution to continue on some way beyond the predicted branch, even though we do not yet know if that is the correct course of action at that point.

This means that any instructions executed after that point are **speculative**, and they must be capable of being discarded and their effects reversed if the branch turns out to be mis-predicted. It follows that the gains here are much larger than simply predicting a branch, and the penalties for **mis-speculation** are larger too. As this topic requires quite a lot more knowledge of the internal architecture of the CPU in question, we will not delve any deeper in this section.

3.4 CPU internal architecture

So far we have examined CPU's more or less as self-contained modules, and we have understood some of the capabilities and limitations of processors in general use. Although a full understanding of the internals of modern processors is a major study effort in its own right, we will here try to establish some basic ideas to build upon at a later date.

We often use the term **microarchitecture** to refer to the internal organisation of a processor. This refers to the internal components of the processor, as far as can be described at a level that is relevant to the programmer, in order to understand how the processor operates and how

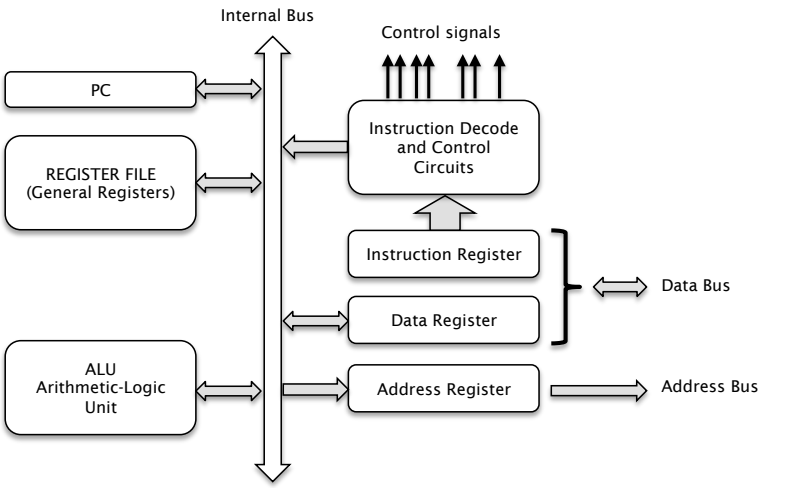


Figure 3.3: Simplified microarchitecture of a CPU. Showing internal components of a processor, including program counter, instruction, data and address registers, arithmetic logic unit, register file, and instruction decoder.

it can be programmed at the machine level. Figure 3.3 shows a typical, albeit simplified, microarchitecture.

Now let us attempt to understand how this microarchitecture operates: We already know that a CPU fetches instructions from memory, and these instructions tell the CPU to perform an arithmetic or logical operation on data held in registers. The **arithmetic logic unit** (**ALU**) does the actual computation.

Registers are simply binary storage elements that can hold numbers of a certain size. We often talk about a register being 8-bit, 16-bit, 32-bit, and so on. What we mean is that a single register can contain a binary number with that quantity of bits.

We can see in Figure 3.3 that there are a number of different registers, and these have different functions:

The **PC register** is also known as the **program counter**, and keeps track of where the next instruction is in memory, so we can fetch it when needed. As we noted earlier, if a branch occurs, the value of the PC may jump to a completely different location in memory. Otherwise it simply increments to the next instruction location in a linear fashion.

The programmer also has access to general registers, which are usually organised into a block known as the **register file**. How many registers are available will vary widely depending upon the processor in question. Some have as few as two, some have hundreds of registers, though 8 or 16 registers is much more typical.

The **data-register**, **address-register**, and **instruction-register** are all needed by the CPU to operate, but are not normally visible to the programmer. Think of them as operating 'behind the scenes', to make sure information gets to the right places at the right times.

When the CPU needs to access a memory location, it will place an **address** in the address register, and then perform a **read** or **write** operation.

If it performs a memory read operation, the resulting value fetched from memory will be placed in the data register, or the instruction register, depending upon the kind of read: an **instruction fetch** reads instructions, a **data fetch** reads data values.

The instruction register holds the current instruction, also known as an **opcode**, whilst the control unit examines it and **decodes** it.

If a memory write operation occurs, then whichever value, previously having been placed in the data register, is transferred to memory. So we can see that registers allow data or instructions to be transferred to and from memory, and to dictate which locations should be accessed.

In order to coordinate these activities, and decide what should happen and when, the CPU has a **control unit**, a digital circuit that is capable of generating sequences of control signals to tell the internal components of the CPU what to do, and when. It operates as follows:

- An instruction is read from memory into the instruction register.
- The instruction is decoded (converted into control signals)
- The control unit provides the correct sequence of control signals during one or more clock cycles.

Exactly what control signals are generated will depend upon the instruction being decoded. There can be hundreds of control signals in a modern processor, so instruction decoding can be quite a major design requirement.

One of the things that the control unit is responsible for is to tell the ALU (arithmetic-logic Unit) what kind of operation to perform on data. The ALU operates as follows:

- The **Control Unit** sends one or two register contents to the **ALU** by performing register read operations from the source registers.
- The **ALU** performs the chosen operation
- The result is sent back to a destination register by a **register write** operation, sometimes referred to as a **register write-back** operation.

ADD R0, R1, R5 (what the programmer sees) **What the CPU does:**	**CPU Activity**
• **Copy** the **PC register** to the **Address register** • Perform a memory **Read** to the **instruction register**	• **Instruction Fetch**
• **Decode** the instruction in the **instruction register**	• **Instruction Decode**
• **Copy** Register 0 to the **ALU** • **Copy** Register 1 to the **ALU**	• **Register Read** • **Register Read**
• Tell the ALU to **ADD**	• **Execute**
• **Write Back** the ALU result to **register** 5	• **Register Write**

Figure 3.4: A typical micro-instruction sequence. Shows the key stages of an instruction life-cycle: instruction fetch, instruction decode, register reads, execute phase, and register write-back.

In a straightforward microarchitecture, any register can be read or written by the control unit. The control unit can also generate **constant data values** and send these to the ALU or registers. What is also quite clear is that even the simplest instruction visible to the programmer causes a potentially complex sequence of operations to be performed within the machine. As a result of this, it is quite possible that a single instruction might take several clock cycles to be completed, hence a CPI of 2, 3 or more might be observed.

3.4.1 An instruction 'in flight'

We sometimes refer to instructions as being **in flight**. What we mean is that it is in progress and being processed through its various stages of life. Let us take a very simple example. We wish to add two registers together, and this sequence of events is represented by Figure 3.4.

So we can see that any single instruction from the programmer's point of view, is actually a sequence of **micro-instructions** as far as the CPU is concerned internally. So programs are made of instructions, and instructions are made of micro-instructions.

We can also see that we can perform more complex operations. For example, we could add two register contents together, send the result to the address register, perform a memory read, and then write the result to a third register. This would be performing something rather complex, but also very useful. This particular scenario is used to access elements of arrays or data tables efficiently. We could devise micro-instruction sequences for quite a few cases in this way, which is exactly what the CPU designer has to do, and in the case of CISC processors is taken to the full range of possibilities. The designer then needs to design a control

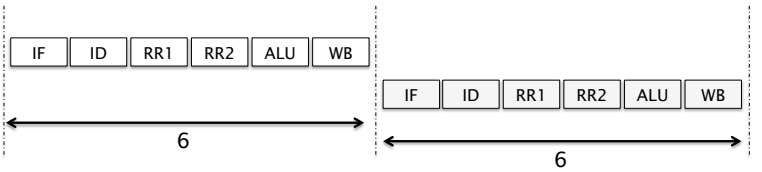


Figure 3.5: Non-Pipelined Instruction Microsequence. Showing IF: instruction fetch, RR1: register read 1, RR2: register read 2, ALU: ALU operation, WB: register write-back.

unit circuit that generates the right internal signals at the right times to make those **microsequences** happen[34].

[34] For those familiar with digital logic design, or wanting to know more, a control unit is often just a variation of something known as a state machine. There are several techniques for implementing them, including Mealy and Moore models, as well as ROM-based designs.

So how many clock cycles does our ADD instruction take? According to our micro-instruction sequence, there are six steps, and this might be envisaged to require at least six clock cycles, implying a CPI of 6.0, which is quite slow. We can see this visualised as a time-line in Figure 3.5, which shows the same instruction repeating.

However, we might observe, from our micro-instruction sequence, that the first two steps are related to what we have referred to previously as instruction fetch and decode. As we noted earlier, it is possible with careful control unit design to overlap instruction fetch, decode, and instruction execution activities. Actually, this also applies to register read and write-backs. It is also possible to read two registers and also write a third register all at the same time, using a circuit known as a multi-port register file.

This being the case, the ADD instruction could appear to take as little as one clock cycle, and CPI=1.0, because almost all activities for an instruction can overlap with a preceding instruction. As Figure 3.6 shows, a new instruction can complete after every clock cycle using the overlapping model. We can see in this diagram that this is possible because different things are happening in different time slots, and they 'zip' together like a zipper quite neatly in this example.

One thing we can observe in Figure 3.6 however, is that by the time the third instruction is overlapping with the first two, we reach a point where its two register reads overlap with the register write-back of instruction one. This is now a potential data-hazard. There are two possibilities here: either delay the register reads to avoid any preceding write-backs, and lose some performance, or make sure that none of the registers being read are the same as the one being written back. This is the basis of an optimisation technique called instruction scheduling.

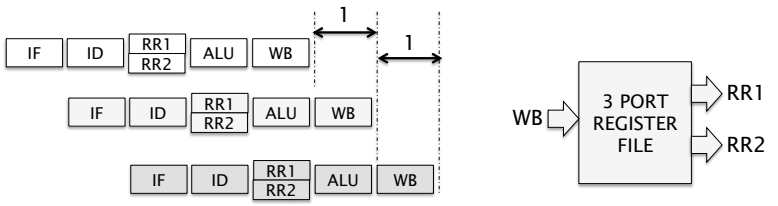


Figure 3.6: Pipelined Instruction Microsequence, showing IF: instruction fetch, RR1: register read 1, RR2: register read 2, ALU: ALU operation, WB: register writeback. To the far right: a 3-port register file, allowing two reads and one writeback simultaneously.

3.4.2 A more advanced microarchitecture

Now that we have established some basic understanding of microarchitecture, we should in theory be able to look at one of the many processor architecture diagrams available from manufacturers that describe the internals of the machine, and understand at least the basic aspects of what is going on.

Now let us look at a slightly more advanced processor model, such as the one in Figure 3.7.

How does this differ from the previous case?

Microarchitecture Differences

- There are now **two ALUs**, and these have been re-designated as integer ALUs.
- A **third Arithmetic unit** has been added, but this one is **floating point**, so it can work with decimal values.
- We have added a **branch prediction unit**, which can work out what the next value of the PC will be, based upon a **branch prediction algorithm**.
- Finally, because there are more ALU's and other units, **the Register File has more connections**.

A significant difference here is that this new microarchitecture can conceivably execute up to four instructions at a time. Of course they cannot be just *any* arbitrary combination of instructions. This processor can execute two integer ALU operations, alongside one floating point operation, and also process a branch instruction by predicting its behaviour.

This can only happen when the combination of instructions is just right. We refer to this as **peak execution performance** or **peak mips**. In reality, due to the way programs are structured, it is more likely that we

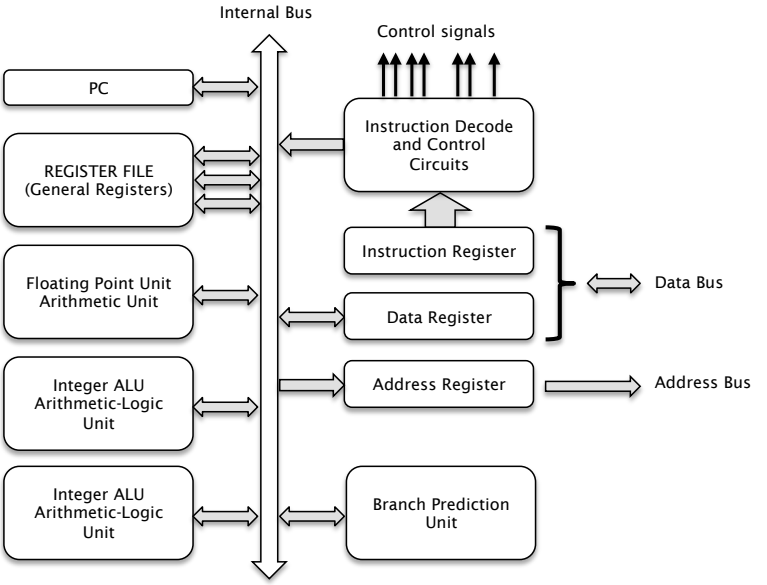


Figure 3.7: A Superscalar Microarchitecture. Showing internal components, including multiple integer ALU's, a floating-point ALU, branch prediction unit, instruction decode, register file, program counter, and additional registers.

will achieve perhaps 3 instructions per clock cycle on average, but this is still a desirable CPI of 0.33 nonetheless. This is an important point to remember: in computer architecture and systems, four times the potential does not always deliver four times the actual benefit. Additionally, this extra complexity is costly in terms of hardware, and may result in lower clock frequencies, requiring more complex components such as that shown in Figure 3.8. There is no such thing as a 'free lunch' in computer architecture.

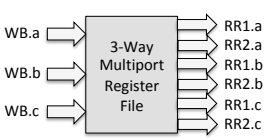


Figure 3.8: 3-way multiport. Given, for example, a three-way superscalar processor microarchitecture, we must then have a three-way multiport register file for peak performance. Such a component would require six read ports and three write-back ports, all capable of operating simultaneously. This implies some quite complex and power-hungry circuitry, and register files are well known as a challenging aspect of modern CPU chip design.

3.5 A little bit of programming

We should now have a good understanding of the general internal behaviour of a relatively simple processor. At this point it should be possible to start to write simple programs, assuming of course that we have a list of instructions, an **instruction set**, that we can understand and use. Since our processor examples were just imagined for the purpose of learning the basic principles, we have the advantage of being able to imagine an instruction set as well.

Of course for these instructions to be valid, they must be capable of being performed using the microarchitecture. We will not worry about this right now, but a good exercise would be to look at each instruction

Table 3.3: Instruction reference sheet.

INSTRUCTION SET REFERENCE SHEET	
Instruction form	**Function and description**
ADD Rx, Ry, Rz	Add Register x and register y together and store the result in register z.
SUB Rx,Ry, Rz	Subtract register x from register y, and put the result in register z.
MUL Rx,Ry, Rz	Multiply register x by register y, and put the result in register z.
DIV Rx,Ry, Rz	Divide register x by register y, and put the result in register z.
LDI Rx, nn	Put the number nn in register x
LOAD [Rx], Ry	Copy contents of memory location pointed to by Rx into register Ry

definition and test it against the microarchitecture to check that they are actually valid.

Let us suppose that the instructions are available as given in our 'instruction reference sheet' given in Table 3.3 and the aim is to write a program to get three numbers from memory and calculate their average, as shown in Table 3.4. In order to make the program behaviour more understandable, we sometimes use a technique called a **program trace**, as shown in Figure 3.5. This shows the step-by-step effects on the processor of each instruction as it is executed, including the program counter and any registers modified. We can see that if we assume the initial memory contents are 6, 8 and 3, then the average of these numbers is 5 (excluding any decimal remainder, since the arithmetic instructions are integer operations).

What we can see here is that the process of writing programs in the lowest level machine code (also known as machine language), that which the CPU understands, is somewhat tedious and long-winded. This is why the vast majority of programs are written in high-level languages such as Java, Python, and C. By using high-level languages, the programmer can focus on what they want to do at a high level of abstraction and not worry so much about what the CPU does at the machine level. A software tool called a compiler or an interpreter then does the rest.

We will not delve any deeper into programming at the machine-level at this point. This is another of those topics that can fill a whole textbook

Table 3.4: Example program.

PROGRAM		(data values in hex)
LDI	**R7, 1000**	Load the integer value 1000 in R7
LOAD	**[R7] , R0**	Load a value from the address held by R7 into R0
LDI	**R7, 1001**	Load the integer value 1001 into R7
LOAD	**[R7] , R1**	Load a value from the address held by R7 into R1
ADD	**R0 , R1, R0**	ADD R0 and R1 together, put result in Register zero
LDI	**R7, 1002**	Load R7 with Integer 1002
LOAD	**[R7] , R1**	Load a value from the address held by R7 into R1
ADD	**R0, R1, R0**	Add R1 onto the R0 and store back n R0
LDI	**R2, 0003**	Load the integer value 3 into R2
DIV	**R0, R2, R0**	Divide value in R0 by value in R2, put the result in R0

Table 3.5: Program Trace. This shows the state of the processes step by step, including the program counter, the opcode and data values it fetches from memory, the equivalent instruction, and the register content written back at each stage. We have assumed we know the contents of memory addresses 'M1000' 'M1001' and 'M1002'. Note how the addresses are represented by two bytes in this table. Results 0Eh and 11h are equivalent to 14_{10} and 17_{10} (decimal), respectively. So 17 divided by 3 gives 5 (as the result is an integer with no rounding capability).

Assumptions: M1000 = 6, M1001=8, M1002=3

PC	OPCODE	DATA	DATA	PROGRAM CODE		REGISTERS
0000	**LDI**	10	00	LDI	R7, 1000	R7=1000
0003	**LOAD**			LOAD	[R7] , R0	R0=6
0004	**LDI**	10	01	LDI	R7, 1001	R7=1001
0007	**LOAD**			LOAD	[R7] , R1	R1=8
0008	**ADD**			ADD	R0 , R1, R0	R0=14
0009	**LDI**	10	02	LDI	R7, 1002	R7=1002
000C	**LOAD**			LOAD	[R7] , R1	R1=3
000D	**ADD**			ADD	R0, R1, R0	R0=17
000E	**LDI**	00	03	LDI	R2, 3	R2=3
0011	**DIV**			DIV	R0, R2, R0	**R0=5**

in its own right, and there are plenty of resources available to study this subject for the more inquisitive reader.

3.6 Extending the Instruction Set

In the preceding sections we have considered some very fundamental and essential machine-code instructions. Of course there are many more possibilities and we could add numerous instructions to those already defined.

A major limitation of the existing instruction set, as presented in Table 3.3, is that it allows only unconditional program execution. In other words, we can specify any number of program steps, but they are executed strictly one after another. Another way of describing this is a **linear execution path**. In order for programs to achieve complex behaviours, there needs to be the ability to execute instructions conditionally. We consider this the essence of 'decision making' within program algorithms, and elements such as loops and 'if-then-else' programming structures would be impossible otherwise. To achieve **conditional execution** we would need to be able to test the state of registers, and then jump to another part of the program depending upon the result. The use of **CMP** (compare) and **JMP** (Jump) instructions is therefore an almost universal expectation for an instruction set. We explore this a little further in Tutorial **??**.

Another reason to include additional instructions is improved code efficiency. Consider for example: if we had a new instruction **INC Rx**, which simply increments the specified register by the value of one each time it is executed. Could we then rewrite our program to use it? Do you think it would be more efficient? Would it use less instructions ?

Importantly, although there are many such cases, where instruction set additions can improve the optimisation and/or utility of the code, the addition of extra instructions is not cost free. Remember that instructions require circuits, and more instructions potentially require more circuitry. The result may be a processor with more instructions, but where each instruction is slower, rather than a lean processor that is fast but has limited instruction variety. This is the root of the well known RISC versus CISC computer architecture trade-off (see Section 2.5.1). Consequently, the balance between important and desirable is a fine judgement in CPU design.

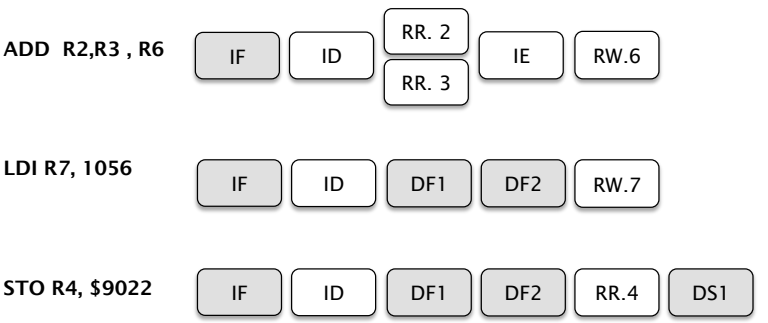


Figure 3.9: Variations of instruction microsequences. Three examples are given here, with memory events shaded. The topmost case is an **add** instruction, which we have already encountered, and which operates on registers only. The middle case presents a memory **load** instruction, which needs to fetch data values from memory (the **data fetch** or **DF** stages). And, finally, the bottom case shows a **store** instruction, which both fetches data (DF) and also transfers data to memory using **data store** (**DS**) stages. Here we assume DF and DS events are single byte operations.

3.7 More on instruction microsequences

So far we have seen some simpler cases of microsequences, where instructions are fetched from memory, and data is manipulated within the register file. However, this is not the only possibility. Indeed, there are many possible variations of instruction microsequences. A few extra examples are shown in Figure 3.9.

It can be observed in Figure 3.9 that three instruction microsequences are presented. An **add** instruction (using only registers), a **load** instruction (which fetches data from memory), and a **store** instruction (which sends data to a memory address).[35] We can see that all three instructions require an instruction fetch **IF** stage, but two of the instructions also use memory for data fetch operations (**DF**), and one also uses memory for a data store (**DS**). This means that memory is needed for both instructions and data values, but ideally at different times. Note that these examples assume that each data fetch or store cycle is a single byte read or write.

[35] Here we can also see that in order to store data at the given address, that address must first be fetched from memory first via DF stages. So, in this case, we have a microsequence that uses three different kinds of memory access within the same one instruction.

It could also be noted that, until instruction decode **ID** has been completed, we cannot start to perform any **DF** or **DS** operations. It is only *after* decoding the instruction that the microarchitecture knows what is required. Because of the need for both instruction and data accesses to use memory, you may be able to see that pipelining of these more complex instructions is also a more complex task in itself. Indeed, we

cannot have an instruction fetch and a data access going to the memory bus at the same time, a principle we know well as the **von Neumann bottleneck**. You will be able to see this if you attempt to sketch out a few of these instructions in random orders, and try and pipeline them as we did with earlier examples.

3.8 Summary

In this chapter we have explored some key concepts relating to processors, primarily the considerations relating to the concept of a CPU. As we shall see later, a processor, as we commonly know it, may well combine additional components onto the same chip to create a more highly integrated circuit system.

We also discovered that a CPU is responsible for fetching program instructions and executing them, either one at a time or sometimes in small groups, and that this can be exploited to gain higher performance. We also found that there are many ways to measure a CPU's behaviour and performance. Circuit complexity, driven by Moore's Law and the ever increasing miniaturisation of transistors, has pushed us toward higher clock rates, higher power density and higher throughput per CPU, but at a significant penalty: heat and power consumption. When it comes to computer systems, and processors in particular, our options for achieving the best outcome with the least resource are always a key concern.

3.9 Terminology introduced in this chapter

Address register	Fetch-execute cycle
Mips	ALU
Floating point	Mis-speculation
Arithmetic Logic Unit	Heatsink
Moores Law	Benchmark
Hit-rate	Multicore
Branch	In-order issue
Non linear	Branch misprediction penalty
Instruction	Opcode
Branch prediction	Instruction issue
Out-of-order completion	Clocks per instruction
Instruction register	Overlapped instruction execution
Cost-benefit	Instruction set
CPI	Integer
Data hazard	Integrated circuit
Data register	Issue width
Data-flow analysis	Jump
Decision point	Kilowatt
Destination register	Linear program sequence
Dynamic branch prediction	Load
Execute	Logic chip
Feature size	Logical operation
Machine code	

These terms are defined in the glossary, Appendix A.1.