Computer Science 382

Lecture Notes

Computer Architecture and Organization

Shudong Hao

August 31, 2022

Computer Science 382

Disclaimer
The lecture notes have not been subjected to the usual scrutiny reserved for formal publications. They may not be distributed outside this class without the permission of the Instructor.

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
https://github.com/fmarotta/kaobook

Edition History
1st edition: August 2022.

Contents

List of Figures

List of Tables

Fundamentals 1

1.1 Number Representations

First thing we need to understand is everything on our computers that we are so used to — graphics, documents, windows, *etc.* — comes down to something physical, *i.e.*, digital signals. Interesting though, digital signals are so simple: they can only transfer currents. How can we interpret those signals for our use then?

One idea is to take the voltage as numbers. If at a certain moment the voltage is very high, we consider that it's expressing a number 1; if later it drops to a very low number, we interpret it as number 0.

One problem though — the voltage is not stable due to materials of the wire, temperature, and so on. It can stay *roughly* at a certain level with small up-and-downs, so we can use a threshold instead. Figure 1.1 shows this idea.

With this in mind, let's look at number representations.

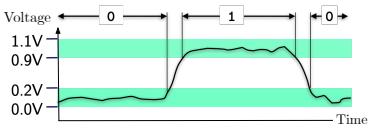


Figure 1.1: How a discrete value can be extracted from continuous signals. In this example, we can get a binary number `010` based on the current change on a wire.

1.1.1 Notations

The integer number system we're most familiar with is decimal. In our course, we will also focus on other systems as well, mostly binary and hexadecimal.

When writing a decimal number, we usually add a letter `D` at the end, *e.g.*, `100D`, `-382D`. The letter `B` will be used as suffix or prefix for binary numbers: `100B`, `100010B`, `0b1010`, *etc.* For hexadecimal, we can add `H` at the end such as `1AB5H`, but we can also add `0x` as prefix: `0x1AB5`. In the future, without specifying which number we're using, you should be able to recognize them correctly based on suffix or prefix.

1.1.2 Binary Numbers

Binary numbers are also sometimes referred as "binary patterns" or "binary sequences". Binaries simply consist of digit 0s and 1s. Each digit is also called a **bit**. When we write a binary number such as `100101001000`, the left-most bit is called **most significant bit (MSB)**, while the right-most **least significant bit (LSB)**. [1]

Because machines usually store multiple bits together, there are also some binary-related sizes or units we need to know about.

A **byte** is always 8-bits, and this is the unit we're going to use most frequently. [2] When we have more bytes, the unit definition could vary among

1: Later we'll talk about most/least significant bytes, whose acronyms are also MSB and LSB. Without specifying, either B is for bit or byte should be clear based on context.

2: Fun fact, 4-bits is called a **nibble**. We rarely use that though.

different machines. On a 64-bit machine (which is the most popular these days), a **word** has 32 bits (or 4 bytes), while a **half word** has 16 bits (or 2 bytes). 64-bit is a **double word**, or **quadword** (for historical reasons).

We can go larger for sure, but when there are too many bytes, we use another system, the IEC prefixing system, which is similar to scientific notation but with powers of 2:

Kilobyte (KB)	$= 2^{10}$ bytes	Megabyte (MB)	$= 2^{20}$ bytes
Gigabyte (GB)	$= 2^{30}$ bytes	Terabyte (TB)	$= 2^{40}$ bytes
Petabyte (PB)	$= 2^{50}$ bytes	Exabyte (EB)	$= 2^{60}$ bytes
Zettabyte (ZB)	$= 2^{70}$ bytes	Yottabyte (YB)	$= 2^{80}$ bytes

1.1.3 Binaries and Decimals

Recall that all numbers are represented as binary numbers, which can be directly mapped to the change of currents on a wire. We as humans, however, rarely use binary numbers, and we're more familiar with decimal numbers. Therefore, the conversion between binaries and decimals are inevitable.

3: We only talk about integers here, not floating points.

Before we start, think how we represent decimal numbers. [3] We can have a positive number such as $+382$, but also a negative one like -220. In machines, how can we represent the pus and minus sign?

Let's start with the unsigned conversion, which is the simplest case, and you're probably already familiar with it.

1.1.3.1 Unsigned Integers

▶ **Conversion to Decimals.**
Given a binary number of d bits $r_{d-1}r_{d-2}\ r_1r_0$, the decimal number is calculated as follows:

$$U = r_{d-1} \cdot 2^{d-1} + r_{d-2} \cdot 2^{d-2} + \ + r_1 \cdot 2^1 + r_0 \cdot 2^0 \tag{1.1}$$

$$= \sum_{i=0}^{d-1} r_i \cdot 2^i. \tag{1.2}$$

You can see that all digits of the binary number are used to convert into the decimal number.

▶ **Extension.**
Later in this course we'll usually need to extend a binary number to a larger size, *e.g.*, extend a 8-bit number to a 16-bit number while keeping its value the same. For unsigned numbers, we can just do **zero extension**, because no matter how many zeros we add to the front, in Equation (1.1) they don't carry any weight eventually. See Figure 1.2 for an example.

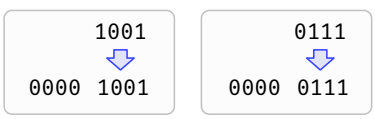


Figure 1.2: Zero extension. In this example, we want to extend unsigned binary numbers of four bits to bytes.

▶ **Data Range.**
When it comes to binary, another thing we usually talk about is the range of the number. Let U^d_{max} and U^d_{min} denote the maximal and minimal number an unsigned integer of d bits can represent, then we have:

$$U^d_{max} = 2^{d-1} - 1 \tag{1.3}$$
$$U^d_{min} = 0. \tag{1.4}$$

1.1.3.2 Signed Integers

Signed numbers are not very complicated either. The only difference is we use the MSB as the sign: 1 for negative, and 0 for positive.

▶ **Conversion to Decimals.**
When converting a signed binary into a decimal, we simply make the weight of the MSB negative:

$$S = -r_{d-1} \cdot 2^{d-1} + r_{d-2} \cdot 2^{d-2} + \quad + r_1 \cdot 2^1 + r_0 \cdot 2^0 \tag{1.5}$$
$$= -r_{d-1} \cdot 2^{d-1} + \sum_{i=0}^{d-2} r_i \cdot 2^i. \tag{1.6}$$

The representation of signed number is called **two's complement**. [4] So... does `1000b` represent -0??

4: There's also one's complement if you're interested.

▶ **Signed Extension.**
Now that we're dealing with signed numbers whose MSB is a sign, when we extend the numbers we cannot simply add zeros to the front. For example, if we want to extend `1010b` to a byte with zero extension, we'll get `00001010b`, which is +10, apparently a different number.

But what if we just do zero extension and make the highest bit the sign, as in `10001010b`? Do the conversion using Equation (1.5) and you'll see that's not right either.

We're getting so close though. `10001010b` is -118, and our target number is -6. If we can add +112 to `10001010b`, we'll get the correct number.

With a bit guess, we notice that in the number `10001010b`, the three contiguous zeros take exactly a weight of +112, then why don't we just make them all 1s, as in `11111010b`? This is exactly the **signed extension**!

Formally, for a signed extension, if the original number's MSB is 1, we just pad all 1s to the front; otherwise all 0s. This way, we compensate the negative increase of the weight due to the extension. See Figure 1.3 for an example.

☢ Caution!

A common mistake is simply take MSB as a sign and apply Equation (1.1) to the rest of the bits. For example, `1010b` is -6 in decimal based on our equation above. However, if you just take the leading 1 as negative sign, and convert `010b` using Equation (1.1), you'll get -2, which is obviously wrong.

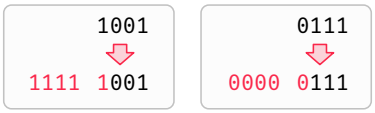


Figure 1.3: Signed extension.

▶ **Data Range.**
Let S^d_{max} and S^d_{min} the maximal and minimal numbers a signed d-bit binary can represent, respectively:

$$S^d_{max} = U^{d-1}_{max} = 2^{d-2} - 1, \tag{1.7}$$
$$S^d_{min} = -2^{d-2}. \tag{1.8}$$

Figure 1.4 shows a mapping from signed number to unsigned number.

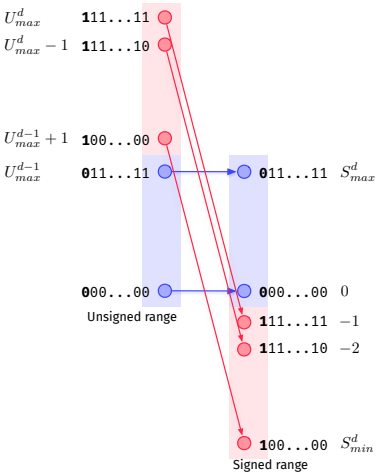


Figure 1.4: For non-negative numbers, signed and unsigned have no difference. When consider MSB as sign bit, the same binary pattern will be mapped to a positive number in the unsigned range.

In fact, there's a small trick to convert decimals to two's complements quickly. Say we want to know the 4-bit two's complement of decimal -5. First, we get the unsigned binary representation for its absolute value 5, which is `0b0101`. Then flip every bit to `0b1010` and add 1 to it, which gives us `0b1011`, and that's exactly the two's complement of -5. If it's a positive number, then the two's complement is identical to the unsigned binary. [5]

5: Can you think of a justification why this trick works?

1.1.4 Binaries and Hexadecimals

One nice thing about conversion between hexadecimals and binaries is they can be mapped directly: every four-bit binary is corresponding to a hexadecimal digit:

```
1010 → A   1011 → B   1100 → C
1101 → D   1110 → E   1111 → F
```

Thus, if you want to convert a binary `000101101010011111010100` to hexadecimal, you just need to chop them into four-bit chunks:

```
0001  0110  1010  0111  1101  0100
```

and convert every four-bit into a digit mapped in hexadecimal:

```
0001  0110  1010  0111  1101  0100
  ↓     ↓     ↓     ↓     ↓     ↓
  1     6     A     1     D     4
```

Then we get the hexadecimal number `0x16A7D4`.

Given a signed hexadecimal number, if the leading digit is greater than 7, we say the number has to be negative. Can you explain why?

Quick Check 1.1

1. For each of the following statements, state if it's true or false, and explain why it's true or false:
   a) Depending on the context, the same sequence of bits may represent different things;
   b) If you interpret a N bit two's complement number as an

unsigned number, negative numbers would be smaller than positive numbers.

2. For the following questions, assume an 8-bit integer and answer each one for the case of an unsigned number and two's complement number. Indicate if it cannot be answered with a specific representation:
   a) What is the largest integer? What is the result of adding one to that number?
   b) How would you represent the numbers 0, 1, and -1?
   c) How would you represent 17 and -17?
3. What is the least number of bits needed to represent the following ranges using any number representation scheme:
   a) 0 to 256;
   b) -7 to 56;
   c) 64 to 127 and -64 to -127;
   d) Address every byte of a 12 TB chunk of memory.

1.1.5 Binary Operations

Calculations between binary numbers are **bit-wise operations**, and it's not that different than decimal calculations. In computer systems, however, each binary has a fixed width, *e.g.*, 64 bits, so the result might be different or not make sense mathematically.

1.1.5.1 Fixed Width Binary Arithmetics

Binary arithmetics such as addition, subtraction, multiplication, and division follow the same rules as decimal numbers. We only look at addition here, but the situations we discussed here apply for the other three calculations as well.

Let's look at an example first, where we restrict both operands and the result to be only one byte. Assume operand `a` is `0b1111 1111`, while `b` is `0b0000 0001`. When we perform `a + b` by hand, we know we need to carry one bit of `1` to the front, so we have `a + b = 0b1 0000 0000`. However, remember we want to restrict the result to only one bytes, we can only save the lowest eight bits, which makes is `a + b = 0b0000 0000`:

```
      1111 1111
+     0000 0001
-------------------
  1   0000 0000
```

In this case, when the result has a carry (or a borrow in subtraction), we say there's a **carry** happened. When we interpret the operands as *unsigned* integers, we see this results in `255 + 1 == 0`.

Now, what if we interpret the binary numbers as *singed* integers? Let's look at the following example:

```
      0111 1111
+     0111 1111
  ---------------
   1  1111 1110
```

This time, we add two `0b0111 1111` together, and get `0b1111 1110`. If we interpret them as signed numbers, this is basically `127+127 == -2`. Now this doesn't make sense mathematically — how come adding two positive numbers results in a negative number? In this case, we say there's an **overflow**.

Assume the lengths of the operands are d bits, and when we perform addition on them we result in a $d+1$-bit number, so we have to get rid of MSB. As shown in Figure 1.5, if the result is between -2^{d-1} and $2^{d-1}-1$ (inclusive), truncating MSB doesn't affect our result. However, if it's between $[-2^{d-1}-1, -2^d]$, truncating MSB will take us to positive range. This is called **negative overflow**. Similarly, truncating a number in the range of $[2^{d-1}, 2^d-1]$ we end up having a **positive overflow**. Note that overflow is meaningless when we interpret the numbers as unsigned.

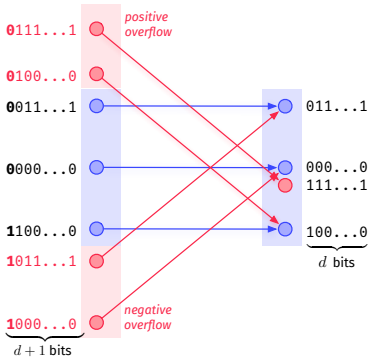


Figure 1.5: Truncating MSB of a $d+1$-bit binary will result in positive or negative overflow. There's no effect on any number between $[-2^{d-1}, 2^{d-1}-1]$.

Quick Check 1.2

Is the following statement true or false, and why? It is possible to get an overflow error when adding two signed numbers of opposite signs.

1.1.5.2 Shifting

As the name suggests, shifting is simply to shift every bit several digits over. There are two directions: left and right, meaning we can shift the number to the left or the right.

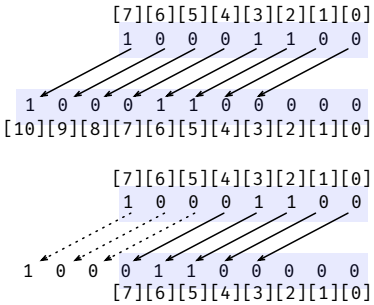


Figure 1.6: We apply three bits of logical shift left operation to binary number `10001100b` (top). If the size of the binary is limited (bottom), the MSBs are discarded.

- **Logical shift left:**
  In logical shift left, we move every bit of the original number to the left, and patch zeros at the end. Figure 1.6 (top) shows an example, where we shift left three bits. One interesting fact about shifting left is, if we shift left `n` bits to a binary number `b`, the result is equivalent to $b \cdot 2^n$. You can verify this: shift left 2 bits on binary `100b` is `10000b`, which is `100b` $\cdot\, 2^2$. This is an important property actually. Multiplication is a complicated operation, hardware-wise, and is very slow. However, shifting is fast and simple. Later in this course you'll see lots of places where we use shifting instead of multiplication.

  When we restrict the length of the binary result, as in Figure 1.6 bottom, we will discard the most significant bits that are out of the binary size.

- **Logical shift right:**
  Shifting right needs to be careful, since MSB can be a sign bit. If we apply the same rule as in logical shift left, we have logical shift right, where shift every bit to the right, discard least significant bits, and pad zeros to the beginning. It's simply the opposite of logical shift left. A similar example is shown in Figure 1.7.

  Even though the bit-wise operation of logical shift left and right are the opposite, shifting right cannot be regarded as dividing by two. This can be easily verified from the examples shown in Figure 1.7. The most obvious thing to notice is that, if the number is a signed integer, shifting right might even change the sign of the number. This is true when the MSB of the original binary number is 1. We call this "logical" because it's simply operated based on how we move the bits around without thinking about the values of the numbers. To make the shifting right similar to division, we'd need arithmetic shift right operation.

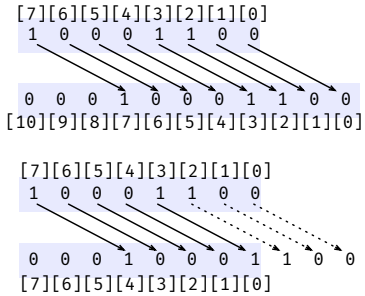


Figure 1.7: We apply three bits of logical shift right operation to binary number `10001100b` (top). If the size of the binary is limited (bottom), the LSBs are discarded.

- **Arithmetic shift right:**
  Arithmetic shifting is similar to logical, except that we pad MSBs with the original numbers MSB. An example is shown in Figure 1.8. If original number's MSB is 1, we pad 1s to the front, to preserve the negative sign of the number. If it's 0, then it's same to logical shift.

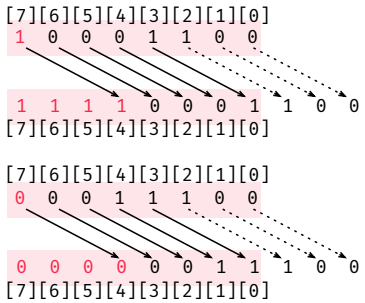


Figure 1.8: In arithmetic shift right, we pad MSBs with copies of original numbers MSB. We only show the versions where the binary size is limited.

Quick Check 1.3

Assume we have a binary number `1100 0101`. How can we apply shifting operations on this number so we can get `0000 0101`? That is, only preserve the lowest four bits and zero out the rest.

1.1.5.3 Bit-wise Logical Operation

Another common type of binary operation is logical operation, such as `and`, `or`, and `xor` (exclusive-or). For this kind of operations, we just need to align the two binary operands and do bit-wise operation on every bit. Same as what you learned in Discrete Structures, `1 and 1` is `1`; `1 and 0` is `0`, *etc*.

One interesting application of logical operations that we'd like to mention is masking. In some situations we'd like to only extract several bits of a number. For example, we have a binary number `11001010`, and we want to extract the 6th bit and zero out all other bits, so that we have `01000000`. We can certainly use masking to achieve this.

Notice both `0 and 0` and `0 and 1` result in `0`, so we can create a binary number where only the bit we want to extract is 1 and all the others are zero. Then we perform bit-wise `and` operation between the original number and this new number we created.

Quick Check 1.4

Assume we have a binary number `1100 0101`. How can we apply only bit-wise logical operations on this number so we can get `0000 0101`? That is, only preserve the lowest four bits and zero out the rest.

1.2 Basic Components of Microprocessors

Most of our modern computers are built on the simple model from John von Neumann, called the **von Neumann model** or **Princeton architecture**. Figure 1.9 shows us a visualization of this model. For our course, we focus on Central Processor Unit (Chapter 3) and Random Access Memory (Chapter 4) most of the time. I/O devices will be mentioned, but not discussed in depth.

1.2.1 Central Processor Unit (CPU)

As its name suggests, CPU is the core of all computers, since it's where the actual calculation happens. Everything we see eventually needs to go through CPU to calculate results. Roughly speaking, the two most important components of CPU are **registers** and **arithmetic logic unit (ALU)**.

1.2.1.1 Registers

Consider a simple expression `2 + 3`: the numbers 2 and 3 are the operands, while the plus sign is the operator. In CPU, the operands need to be stored and ready in registers, so that they can be sent to ALU for calculation.

Most of the machines have a set of registers for us to use, and this set is usually called a **register file**. You can think registers are just small storage to store one binary sequence.

In our course, we'll be using ARMv8 architecture, where we have 32 **general purpose registers**, and each of them can store 64 bits. We call it general purpose, because they can be used by us to store any binary sequence we want. There are also special registers that we cannot modify the numbers stored there, and we'll introduce them later.

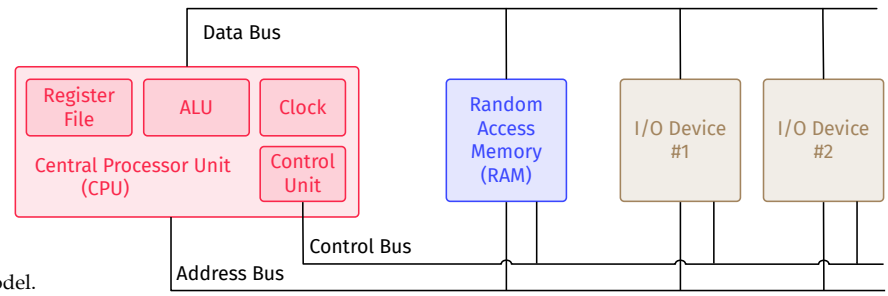


Figure 1.9: A typical von Neumann model.

1.2.1.2 Arithmetic Logic Unit (ALU)

Once the operands are ready in the registers, they'll be sent to ALU for the real calculation. ALU receives two operands, and does simple calculations as specified. In fact, ALU is very simple: the only calculations it can do are arithmetics (addition and subtraction) and bit-wise logics (`and`, `or`, *etc*). Later in this course, we will understand how these simple elementary-school-math in ALU leads to something so magical and powerful.

1.2.2 Random Access Memory (RAM)

Most of our computers are **stored-program computers**, meaning our programs are stored as data inside the computer. Typically, a random access memory, or RAM, is connected to CPU to store them.

- **Data Storage.**
  You can think the RAM as a large array or list of bytes. It's **byte-addressed**, meaning each byte in the memory has its own address. The addresses typically range from 0 to a very large number, depending on the size of the RAM. For example, with a 4GB RAM, we can store $4 \cdot 2^{30}$ bytes, and therefore the address range will be $[0, 2^{32} - 1]$. Apparently, this 4GB memory is never large enough for a modern computer, or in fact much much smaller than what we actually need. Later in this course, we will learn a very genius idea about **virtual memory** and **physical memory**. For now, their difference doesn't really matter.

  Figure 1.10 shows us a visualization of memory storage, where you can see every eight bits, *i.e.*, every byte, has a unique address. As you can see, we usually use hexadecimals to represent addresses. Now with this address, we can get any number we like, or store a number to any address we like.

- **Data Transfer.**
  Note here RAM is **not** inside CPU; thus, if we want to do some calculations using CPU on data from RAM, we need to move the data from RAM to registers first. With Figure 1.9, you probably have guessed that CPU sends an address to RAM through **address bus**, and then RAM will send the data stored at that address back to CPU through **data bus**.

  Buses are just a collection of wires used to transfer data between CPU and other devices, where each wire transfers a single bit of zero or one. Remember in Figure 1.1 we say we can extract a discrete value from continuous change of signals on a wire. That is true of course, but what if we want to transfer one byte? It's a waste of time to use just one wire, because we'd have to wait for 8 seconds (assume each second transfers one bit) to get the number. What makes sense is to group 8 wires together, and transfer the 8 bits on them all

Address	Data
	...
0xFF08	
0xFF07	
0xFF06	
0xFF05	0101 1010
0xFF04	0000 1000
0xFF03	0000 0000
0xFF02	1100 1010
0xFF01	0100 0100

Figure 1.10: Visualization of RAM. Notice each byte (8 bits) has a unique address. The addresses are ranged from `0x00...0` to `0xFF...F`. What's the size of this RAM?

at once, one bit per wire. This group of 8 wires is called a bus.

The width of the buses can vary based on their needs. For 32-bit machines, an address bus has a width 32, and thus it can address at most 4GB of memory. On 64-bit machines, the address bus has a width of 64, which makes the addressable space 2^{64} bytes, or 16EB. The width of control bus depends on how many signals the control unit needs to send to different parts of the machine. We'll learn in this Chapter 3.

1.2.3 Peripheral Devices

CPU and RAM are the main focuse of our course, but for a real computer that's far from complete obviously. First of all, remember the storage of memory is very limited, and therefore we can't store all our data/program into it. In fact, only when we start executing a program will it be loaded into RAM. Also, we need to connect monitors, keyboard, *etc.* We call these peripheral devices, and from Figure 1.9, we see that they are mostly I/O devices, including hard drives.

Quick Check 1.5

Now let's do a simple quick check to make sure you understand everything so far.
True or False?

1. All data stored in CPU has an unique address;
2. Random access memory stores all the files on one's computer;
3. Random access memory is byte-addressed;
4. If we want to do a calculation such as $24-13$, we store the two operands into RAM, and transfer them to ALU to perform the calculation. Once ALU computes the result, it'll be directly transferred back to RAM;
5. In a program written in Java, for example, we declare integers, characters, floating points, *etc.*, for variables. RAM also stores them in the forms of integers, characters, and floating points.

1.3 Computer Abstractions

With the most underlying organization introduced, let's go a bit higher. One of the greatest ideas in computer science is **abstraction**. It's not easy to go from circuits to the fancy stuff we see on our computers, and therefore, from the bottom (hardware), the computer scientists built one layer on top at a time, which eventually transforms digital signals to everything we're familiar today.

In computer science, abstraction means the underlying details are removed or ignored, so that it provides a convenient interface for upper level applications. For example, when you write a Java program and want to print something out, you'd use function `System.out.println()`. Now, you think that's straightforward, right? But remember, when you want to print something to screen, which is physical, you need to manage how the actual hardware manages to bring this string to the screen. However, as a programmer, did you really care about those? No, the only thing you did is to call the `println()` function, and this is because this function is exactly an abstraction of the job you want to do.

Figure 1.11 shows us the layers between the actual hardware (digital logic) and our level (high-level language). You can see this abstraction greatly reduces our work and expedites the development of computer science. Now if we run a C or Java program, we really don't have to think about how different wires or buses transfers 0s or 1s; all we need to do is to focus on the task itself.

Our course is on the level of **Microarchitecture**, so not too low not too high. Each layer in the abstraction is, however, closely coupled, so it's impossible to entirely look at one layer. Therefore, we'll have to look at its neighbors as well. In Chapter 2, we start with instruction set architecture and focus on assembly language to get a hang of how CPUs actually execute programs. Then in Chapter 3, we'll first briefly talk about digital logic, and then start building our microprocessor with different components.

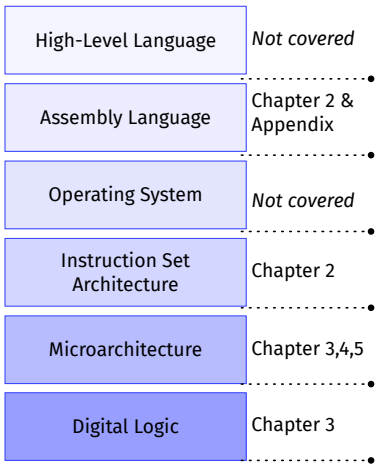


Figure 1.11: Abstractions of computer systems.

1.4 A Peek Into Memory with C

To prepare you for the madness later in this course, let's start from the layer you're most familiar with, which is the high-level language. Not all high-level languages are created equally, though. Our purpose is not to learn a new language; instead we want to use a language that's high-level enough for understanding but also has close connection to low-level hardware design as an entry point. C apparently wins in this perspect.

1.4.1 Quick Start

C programs start with `main()` function, which is the entry point of a C program. The simplest C program looks like this:

```c
int main() {
    return 0;
}
```

Variable and function declarations are very similar to Java, so we will skip that part. One difference is that C is not an object-oriented language, so the functions and variables don't have attributes such as `public`, `private`, *etc.*

1.4.1.1 Data Types

In C, we have several basic data types: `char`, `short int`, `int`, `long int`, `float`, and `double`. The difference among the three types of `int`s is they have different numbers of bits to represent an integer, so they will have different ranges of representation. Same for `float` and `double`, though the result is they have different precisions.

In fact, `char` types are also integers, but can only store eight bits (see Section 1.4.2.2). The integers they store represent a character's ASCII code, so the following two declarations are equivalent:

```
char c = 'A';
char c = 65;
```

because character `'A'` has an ASCII code of 65.

Notice here we don't have string types. Since a string is basically an array of characters, we can declare it using three different methods:

```
char  str[]  = "Hello!";
char* str2   = "Hello!";
char  str3[] = {'H','e','l','l','o','!',0};
```

The difference between the first two is the place where the string is stored, and we can just ignore it now. Notice for the last method, we add a `0` at the end. This is a **null terminator**, marking the end of the string. If we use the first two methods, a null terminator will be automatically attached, so we do not need to add it manually. The null terminator can be declared using integer value of 0, or a character `'\0'`.

Integer data types (including `char`) by default are **signed**. You can add modifier `unsigned` to make them unsigned. For example,

```
int var1 = -253;
unsigned int var2 = -253;
printf("var1 = %d, var2 = %u\n", var1, var2);
```

We'll introduce `printf()` in the next section, but based on the output, you can see they have different values, even if we declare them to be the same. Actually you can verify that, if an integer takes four bytes, -253 is indeed 4294967043 when interpreted as an unsigned integer.

1.4.1.2 Binary Operations

C language can deal with binary numbers as well. We show the operators and their descriptions in Table 1.1.

Notice that what's different than C++ is `<<` and `>>` are not used as stream operators in C, so we cannot print something to the terminal using them. They are simply used for numbers and manipulate bits. Also, `&`

Table 1.1: Binary operators in C language.

Operator	Description
<<	Logical shift left
>>	Arithmetic shift right
&	Bit-wise and
|	Bit-wise or
^	Bit-wise xor
~	Bit-wise negation

and `|` are different than logical operators `&&` and `||`. Please be careful when you use them.

You can certainly apply them to any numbers you like. See the following example:

```
char a = 10;        /* a = 0000 1010b */
char b = 20;        /* b = 0001 0100b */
char c = a & b;     /* c = 0000 0000b */
char d = a << 3;    /* d = 0101 0000b */
```

where left operand of `<<` is the original number, and right operand is the number of bits we want to shift.

Quick Check 1.6

Assume `a = 0b1000 1011`, `b = 0b0011 0101`, and `c = 0b1111 0000`. Calculate the results of the following C statements:

1. `a & b;`
2. `a | c;`
3. `a | 0;`
4. `a | (b >> 5);`
5. `a & (b >> 5);`
6. `~((b | c) & a);`

1.4.1.3 Formatted I/O

A very common thing to do is to print something to the terminal. To use I/O utilities in your C code, you should include the header file as the first line:

```
#include <stdio.h>
```

If it's just a string, such as those declared in the code listing above, we can use `puts()` function:

```
puts(str);
```

If we want to print, say, a variable, we'd need to use **formatted output**, `printf()`. The first argument of `printf()` is a string that represent what the output should look like. For every variable, we'd need to put a "placeholder" there, called **format specifier**. Then the following arguments would be the variables we want to print. For example:

```
printf("Variable a = %d\n", a);
```

where we assume there's a variable `a` whose value is 10. Then the output will be like: `Variable a = 10`, with a new line. You can see that `%d` is replaced by the value of `a` when printed out. Different types require different format specifiers. The common ones are shown in Table 1.2

Table 1.2: Commonly used format specifiers in C.

`%d`	Decimal signed integers
`%u`	Decimal unsigned integers
`%f`	Float numbers
`%lf`	Doubles
`%c`	Characters
`%s`	Null-terminated strings
`%p`	Pointers

In C, we cannot print an entire array out with one single `printf()`; instead, we need to loop into the array, and print each element indivitually:

```
double arr[5] = {3.1, 4.223, 5.152, 10.01 , 2};
for (int i = 0; i < 5; i ++) {
    printf("The %d-th element is %lf\n", i, arr[i]);
}
```

However, the only exception is strings. If we have declared a string, we can just print it out using one `printf()` with `%s`:

```
char str[] = "Hello!";
printf("The string is %s\n", str);
```

1.4.1.4 `goto` Statement

We're fairly familiar with some typical control structures such as loops and `if-else`, so we'll skip them here, but bring a unique and "infamous" keyword in C, called `goto`. We say it's infamous, because it has been criticized for so long.

What it does is very simple: you mark a line of your C code with a label, and you can use `goto` to change your program to execute the line with the label. For example:

```
#include <stdio.h>
int main() {
    int a = 10;
    a = 20;
    goto L1;
    a = 40;
L1: printf("a = %d\n", a);
    return 0;
}
```

In this example, we mark the line with `printf()` using a label `L1`. After we assigned 20 to `a`, we used a `goto` statement to jump to `L1`. As you see, in the print out string, value of `a` is 20, instead of 40, because the statements between `goto` and the destination label `L1` were simply skipped.

Except that you cannot label the statements that are declaring a variable, there's no limit where you can put a label. You can certainly put a label before the `goto` statement like this:

```c
    int a = 10;
L1: a = 20;
    goto L1;
    a = 40;
    printf("a = %d\n", a);
    return 0;
```

But what will happen? You're stuck in an infinite loop! You can even do this:

```c
L1: goto L1;
```

But as you see `goto` statement will cause problems: the program logic and algorithm becomes very unclear, the control is not structured, and it makes the program difficult to debug and understand. That's why it's been criticized so hard widely, and almost everyone advices against using it. Please **do not use `goto` in any of your assignment/lab/exam in this course.**

So why do we introduce it here? Later when we learn assembly language, which is unstructured, this type of jumping between statements using labels is the only way we can control the program flows. In other words, everything in assembly is "`goto`", so it'd be better to know this from a language we're familiar with.

1.4.2 Pointers

Every variable, even our code, stores in memory when executing the program. If it's in memory, apparently they'll be stored as binary numbers, and every byte will have an address (see Figure 1.10). For C language, we can explicitly use those addresses, which is a major difference than the languages you have learned so far. The addresses in C are called **pointers**. Pointers are "pointing" to a variable in our program, and have a type closely related to the type of the variable they point to.

1.4.2.1 Reference and Dereference

Look at the following program:

```c
int main() {
    int  var     = 10;
    int* ptr_var = &var;
    return 0;
}
```

In this example, we declared an integer `var`, and it'll be stored somewhere in the memory when we're running this program. What if we want to know where it's stored? To get a variable's address, we use `&`, the **address-of** operator, in front of it. To store this address into a variable, we first need

to consider the type of `var`. Since it's an integer, we'll declare a pointer of integer, `int*`, to store the address of `var`.

Given a pointer, if we want to get the value stored at the address, we can **dereference** the pointer by using `*` operator:

```
1 int main() {
2     int  var     = 10;
3     int* ptr_var = &var;
4     int  deref   = *ptr_var;
5     return 0;
6 }
```

When running this program, on line 3, we'll have `ptr_var` to store the address of `var`, say `0xffff1000`. Then on line 4, we use `*` to dereference the address, which gives us the value stored there, and assign it back to variable `deref`. Thus, the value of `deref` is 10.

Everything in our program will be stored in the memory when executing, meaning pointers themselves will also be stored just like any other variables in the memory. If they are also in the memory, that means they also have their own addresses. So how do we get the addresses of pointers, or addresses of addresses?

Following the example above, and based on our current discussion, it's not difficult to derive the double pointer:

```
1 int** pp_var = &ptr_var;
```

1.4.2.2 Pointers and Memory

You probably already know it, but different variable types take different numbers of bytes in memory:

`char`	1 byte	`short int`	2 bytes	`int`	4 bytes
`float`	4 bytes	`long int`	8 bytes	`double`	8 bytes
Any type of pointers: 8 bytes					

When we store a variable, its bytes will be stored in continuous bytes. We know that each byte has its own address, if an integer takes 4 bytes, it should have 4 addresses. Why does a pointer have only one address for one variable? Here's the thing: when a variable takes multiple bytes, the address (pointer) of this variable is the lowest address.

For example, assume `var` is an integer, stored in addresses `0x1000`, `0x1001`, `0x1002`, and `0x1003`. In this case, the address (pointer) of `var`, *i.e.*, `&var` is the lowest address which is `0x1000`.

Which leads to another question then. We know different data types can occupy different numbers of bytes. When you dereference a pointer, how does the system know you want a, say integer, or a double? Remember

pointers also have types, *e.g.*, `int*` and `double*`. If you're dereferencing a `int*`, the system will group four contiguous bytes starting from the address indicated by `int*` and interpret it as an integer. Thus, the type of the pointer determines how many bytes it needs to retrieve from the address.

1.4.2.3 Endianness

Almost all the memories are byte-addressed, meaning each byte has a unique address. We also know some data types occupy multiple bytes. The question is, how do you store these different bytes in the memory, *i.e.*, in what order?

Say we have an integer `int var = 366154`, whose hexadecimal is `0x5964A`. Since each integer takes four bytes and each byte has two hexadecimal digits, we can separate the bytes as follows:

```
0x00  0x05  0x96  0x4A
```

Now these four bytes will be stored in the memory when the program is executing, and each byte has its own address. Let's assume the integer's address `&var` is `0x1000`. Question is, which byte has what address?

There are two ways to manage this. First, we can let the lowest byte occupy the lowest address, called **little endian**:

Address	0x1000	0x1001	0x1002	0x1003
Data	0x4A	0x96	0x05	0x00

Of course we can also let the lowest byte occupy the highest address, which is called **big endian**:

Address	0x1000	0x1001	0x1002	0x1003
Data	0x00	0x05	0x96	0x4A

The endianness is specific to machines, not something we can control. Most of the machines we're using are little endian machines, and so in this course without specifying we assume little endianness. You can check the endianness of your machine by running the following program: [6]

```c
#include <stdio.h>
int main() {
    int var = 366154;
    printf("%p: 0x%X\n", &var, var);
    char* p = (char*)&var;
    for (int i = 0; i < sizeof(int); i ++)
        printf("%p: 0x%X\n", p+i, p[i]);
    return 0;
}
```

6: In the code, we first print out the address and the value of `var` using hexadecimal format (`%X`). To view each byte, we convert the pointer of `var` to `char*`, since each `char` takes one byte. Function `sizeof(int)` returns the number of bytes that an `int` takes, so we iterate from the lowest adderess to the highest, and print one byte at a time.

Also notice that regardless of endianness, the address of a variable stays the same. For the above example, both big and little endianness, the address of `var` is `0x1000`.

> **☢ Caution!**
>
> Note that endianness only happens within multi-byte variables. For an array, all the elements are still placed from low address to high address; it's **within each element** that there might be endianness differences among different machines.

1.4.2.4 Arrays and Pointer Arithmetics

When we want to print out the address of a variable, we use `&` operator, *e.g.*, `int* p = &var`. However, to show the address of an array or a string, we do not need to use address-of operator, because the variable name of an array or a string is already the starting address of it:

```c
double arr[5] = {3.1, 4.223, 5.152, 10.01 , 2};
printf("%p\n", arr);
```

In fact, the name of an array is also the address of its first element, *i.e.*, `arr == &arr[0]`.

Another important operation is called **pointer arithmetics**. Given a pointer, we can add or subtract a number from it (called **offset**) to obtain a new address. Following the code segment above, since `arr` is the starting address of the array, we can add the index of an element to `arr` and get the address of that element:

```c
for (int i = 0; i < 5; i ++) {
    printf("%p\n", arr + i);
}
```

When running this program, you'll notice that the addresses on two consecutive lines have eight bytes differences. This is because `arr` is a `double*` and each `double` takes eight bytes, so adding 1 to the pointer will produce an offset of eight bytes. More generally, for a pointer `p` of type `x` and an integer `i`, when we do `p + i`, the new address will be actually `i * sizeof(x)` bytes higher than `p`.

We have also learned deferencing a pointer to get the value stored at that address, so here's another way to print all elements out:

```c
for (int i = 0; i < 5; i ++) {
    printf("%lf\n", *(arr + i));
}
```

What we are doing here is first calculate the address of the `i`-th element by `arr + i`. Then we dereference the address (pointer) by `*(arr + i)`. Table 1.3 summarizes the equivalency between two methods so far related to arrays.

Table 1.3: Array access methods. The two methods on each row are equivalent.

`arr[0]`	`*arr`
`&arr[0]`	`arr`
`arr[i]`	`*(arr + i)`
`&arr[i]`	`arr + i`

> **Quick Check 1.7**
>
> Can you write a program to show all the bytes in the following array individually, in the order they are stored?

```c
double arr[5] = {3.1, 4.223, 5.152, 10.01 , 2};
```

1.4.2.5 Null-Terminated Strings

In Section 1.4.1.1 we mentioned that strings need to have a null-terminator at the end. A null terminator is simply a `char` whose ASCII value is 0. The reason why we need a null terminator is mainly because of how C handles strings and output.

We know that to print out an array we need to use for loops, where we'll specify the number of iterations we want to repeat. One exception, however, is strings, where we can just use `%s` to print all the characters in the `char` array:

```c
char str[] = "Hello!";
printf("%s", str);
```

Note that `printf()` receives `str`, which is the *starting address* of the `char` array, and so it'll start and continue printing one byte after another. When will it stop, then? That's where the null terminator comes into play: `printf()` will start printing given the starting address of the string, keep printing, until there's a null terminator.

What if we don't have the null terminator? Sometimes it's fine, because there's already no data stored at the end of the string, and therefore its value is just 0, which happens to be the null terminator. But try the following program:

```c
#include <stdio.h>
int main() {
    char a[8] = {'H','e','l','l','o','!','!','!'};
    char b[8] = {'W','o','r','l','d','!','!','!'};
    printf("%s", a);
    return 0;
}
```

And you'll see the output is `Hello!!!World!!!`, which is because `a` is not null terminated. It stops at `b` simply because in memory there's no data stored after the last "`!`", and so by default its value is `0`.

Instruction Set Architecture | 2

In this chapter we will discuss ARMv8 instruction set architecture. From now on, I want you forget all the magical things you learned about programming languages.

2.1 Instruction Format

Instructions consist of four parts and have the following format:

```
1 Label: Mnemonic   Operand_1, Operand_2, ...    // Comments
```

where **label** marks the address of the instruction in memory, which is *optional*; **mnemonic** is the text representation of specific operation we want the instruction to carry out (think about the plus sign in $22 + 33$), which is *required* for any instruction; **operands** are separated by commas, and the amount of operands needed is determined by the actual mnemonic for that instruction (it can vary from 0 to 3). The operands can be numbers (called **immediate**) and/or registers. Lastly, line comments are marked by double slash, and block comments can be used with a pair of `/* ... */`.

2.1.1 Register Names

In ARMv8, we have 32 general purpose registers, each of which can store 64 bits, and one of them is a zero register whose value is always 0 and cannot be changed. Figure 2.1 shows the complete register file in ARM architecture.

Those register names can be used as operands. If `W`-registers are used, only the lowest half 32 bits will be operated on; if `X`-registers are used, all 64 bits will be operated on.

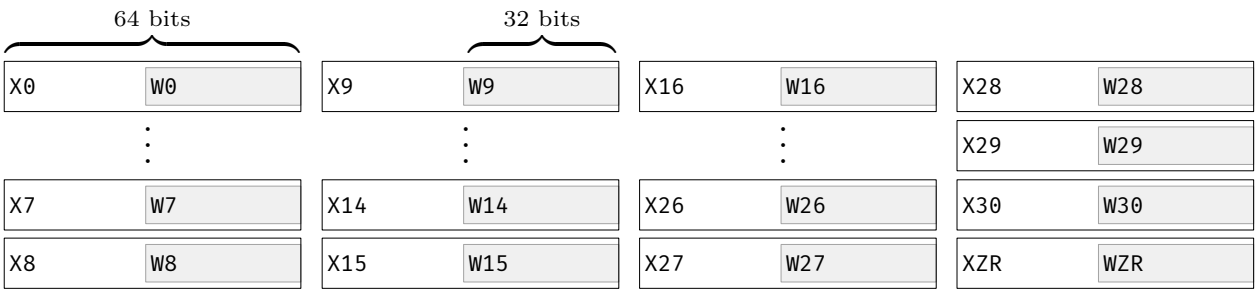


Figure 2.1: Register file of ARM architecture. `X`-registers can store 64 bits. The lowest half 32 bits for each `X`-register can also be used independently as `W`-registers. However, the upper half 32 bits cannot be used independently.

2.2 Accessing Memory

Recall in Section 1.2.1 in Chapter 1, we made it clear that if we want to let ALU to perform a calculation, we have to move the data from RAM to registers first. So in this section, let's see how we can achieve that, and those will be our first assembly instructions! (Exciting!)

2.2.1 Load

To move data from memory to registers, we use **load instructions**: `LDUR`.[1] There are many versions of `LDUR`, but let's start with two most common ones:

- `LDUR Wt, [base,simm9]`:
  loads a word from memory addressed by `base+simm9` to `Wt`;
- `LDUR Xt, [base,simm9]`:
  loads a doubleword from memory addressed by `base+simm9` to `Xt`.

`Xt` represents the destination registers, whereas `base` is a register that stores the **base address** in memory. `simm9` stands for ***s***igned ***imm***ediate ***9***-bit integer, which is used as an **offset** from the base address. It indicates how many bytes from the base address we want to start grabbing data.

1: The instruction stands for ***l***oa***d*** ***u***nscaled ***r***egister. The meaning of the word "unscaled" here doesn't really matter that much for now.

Example 2.1

An integer is currently stored at memory address of `0xFFFFFFFFEB101010`, and this address has already been stored in register `X9`. Please load this integer into the 11-th register.

Solution: First, notice the data we want to load is an integer, which takes four bytes, the size of a word. Therefore, the destination register should be a `W`-register, *e.g.*, `W10`.

Next, remember the address of a variable is its lowest address. Thus, the integer is stored at `0xF..FEB101010` to `0xF..FEB101013`. The starting address `0xF..FEB101010` has already been stored in `X9`, so `X9` is our base.

When we use `W`-registers as destination, `LDUR` will grab four bytes from `base+simm9`. In this example, if we leave `simm9` zero (no offset from base), it'll take the integer back to `W10`. In summary, the instruction should be:

```
LDUR   W10, [X9]
// or LDUR  W10, [X9, 0]
// or LDUR  W10, [X9, #0]
```

The idea of this example is shown in Figure 2.2, where we included another example `LDUR W10,[X9,2]`.

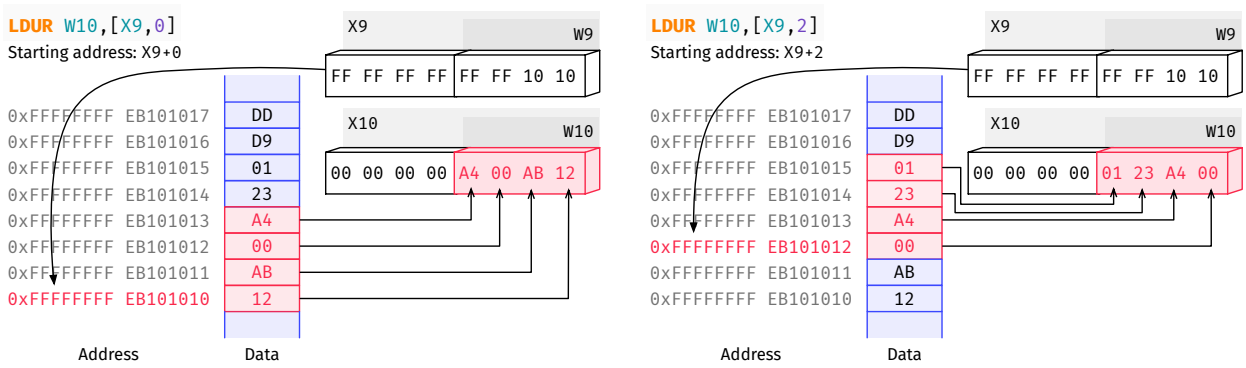


Figure 2.2: `LDUR` will grab a few bytes (determined by the size of the destination register) from the starting address, calculated by `base+simm9`.

> Pay attention to how we move individual bytes into the register: the byte at lowest address is stored in the lowest byte of the register. This is little endian. If the machine is using big endianness, the order is reversed.

We can also load a byte or a half word (2 bytes) into a register, but with caution. Remember the register operands are either `X`-registers (8 bytes) or `W`-registers (4 bytes), so if we want to grab only one or two bytes from memory, we need to also *extend* the data to match the size of the destination. The instructions that can do this for us have the same format for operands as in `LDUR`, but the mnemonic can have variations:

```
LDUR[S]{H|B} {X|W}t, [base, simm9]
```

The `[S]` in the mnemonic is optional; if used, it'll sign extend the data to match the destination register; otherwise it'll zero extend the data. The `{H|B}`, required, stands for either move half-word (`H`) or a byte (`B`). For example, `LDURSH X10, [X9,-24]` will grab two bytes starting from address `X9-24`, and sign extend this two byte data to eight bytes, and store it into `X10`.

2.2.2 Store

Moving data from memory to registers is to load data, whereas moving data from a register to a location in memory is to **store** data. The basic mnemonic for storing data is `STUR` .

Now that the destination is memory which is byte-addressed, we don't need to think about extensions such as byte to double word.

- `STUR Wt, [base,simm9]` :
  stores a word from `Wt` to memory addressed by `base+simm9`;
- `STUR Xt, [base,simm9]` :
  stores a double word from `Xt` to memory addressed by `base+simm9`;

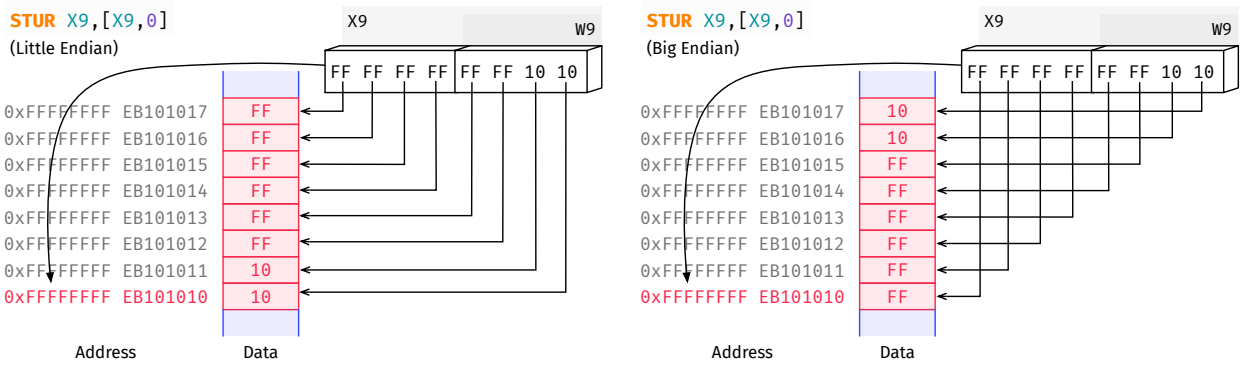


Figure 2.3: STUR is the opposite direction of LDUR. Notice how the same value of X9 is stored differently on little and big endian machines.

- `STURB Wt, [base,simm9]`:
  stores a byte from `Wt` to memory addressed by `base+simm9`;
- `STURH Wt, [base,simm9]`:
  stores a halfword from `Wt` to memory addressed by `base+simm9`.

Remember, assembly is a very simple language; it does exactly what you told it to do. If you want to use `STUR` instruction to store a double word starting at address of `base+simm9`, all the 8 bytes on memory from `base+simm9` to `base+simm9+7` will be overwritten. They won't be saved somewhere.

Example 2.2

Currently `X9` stores a memory address `0xF..FEB101010`, and we want to store this into the memory address indicated by itself, *i.e.*, `0xF..FEB101010`. Write the instruction to achieve this, and draw the memory layout. Assume we're using a little-endian machine.

Solution: The instruction is very straightforward:

```
1 STUR   X9, [X9]
2 // or STUR  X9, [X9, 0]
3 // or STUR  X9, [X9, #0]
```

When drawing memory layout, notice we assume this is a little-endian machine. Therefore, the lowest byte in the register will be stored at the lowest address, which is the base address. We show both little and big endian in Figure 2.3.

Quick Check 2.1

1. **True or False** and why: assume `X0` points to the start of an array called `arr`. `LDUR X1,[X0,3]` will load `arr[3]` to `X1`;
2. Assume we have an array of long integers `long int arr[3]`, and the value of `arr` is stored in register `X9`. Write a sequence of ARMv8 instructions to move all elements in the array from memory to register `X10` and move it back, one at a time.

2.3 Moving Constants & Registers

Later you'll find out that moving data from memory to registers is not always the best choice. Sometimes we just want to move some integer numbers to a register, something as simple as writing an assignment statement in C: `int a = 10`. The instruction we can use is `MOV`.

- `MOV Xn, simm64`: moves 64-bit signed immediate to register `Xn`;
- `MOV Wn, simm32`: moves 32-bit signed immediate to register `Wn`, while **zero out** the highest 32 bytes of `Wn`.

We can also "move" data from one register to another:

- `MOV Xdst, Xsrc`: copies data from register `Xsrc` to `Xdst`;
- `MOV Wdst, Wsrc`: copies data from register `Wsrc` to `Wdst`.

Notice two things here: first, the destination and source registers should match the size; second, it's called "moving", but really it **copies** data from source to destination. This means after the instruction, the data will have two copies: one in the destination, and the other in the source.

For example, `MOV X9,X10` will **copy** the data from `X10` to `X9`. If before `X10` stores an integer value of 382, after the instruction, both `X9` and `X10` will have value of 382. The old value in `X9` will be completely overwritten.

Quick Check 2.2

1. What's the value of `X9` after each of the following instruction in hexadecimal?

```
1 MOV X9, -9
2 MOV W9, 10
```

2. Assume `X9` has a value of `0xFFFF101029AB00FE`, while `X10` has a value of `0x33F59077F2FFABCD`. What are the values of `X9` and `X10` after the following instruction: `MOV W10, W9` ?
3. Which of the following instructions are invalid and why?
   a) `MOV W10, WZR`
   b) `MOV X20, [X22,10]`
   c) `MOV 382, X20`
   d) `MOV XZR, 382`
   e) `MOV 382, 392`

2.4 Data Processing Operations

Once the data have been moved to registers, either from an immediate number or memory, we are ready to perform real computations.

2.4.1 Arithmetic Operations

Let's start with the simplest arithmetics: addition and subtraction. The mnemonics for these two operations are very straightforward: `ADD` and `SUB`.

- `ADD Wd,Wn,Wm` : `Wd = Wn + Wm;`
- `ADD Xd,Xn,Xm` : `Xd = Xn + Xm;`
- `ADD Wd,Wn,simm12` : `Wd = Wn + simm12;`
- `ADD Xd,Xn,simm12` : `Xd = Xn + simm12.`

The subtraction instructions have the same format except the mnemonic is SUB, so we'll skip them here.

When using `W` registers, however, we have to be careful. For example, assume we have `X0 = 0x2222222222222222` and `X1 = 0x3333333333333333`, both full 64-bit numbers. When we use `X` registers to do addition, *e.g.*, `ADD X2,X1,X0`, `X2` will be `0x5555555555555555` without any problem. If we use `ADD W2,W1,W0`, however, the highest 32 bits will be **cleared** (set to zero), and only the lowest 32 bits will be used. Therefore, in this case, `X2` will be `0x0000000055555555`. See Figure 2.4 for an illustration. [2]

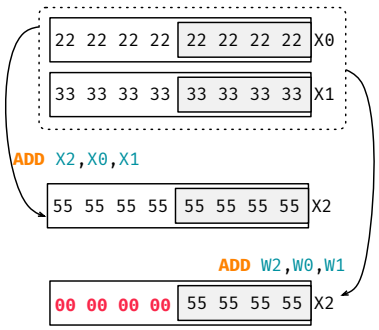


Figure 2.4: Using `W` and `X` registers for addition. If `W` registers are used, the highest 32 bits will be cleared out. The grey boxes are `W` registers.

2: This is very different than other machines such as x86, where operating on low bits of a register does not change the high bits at all. Therefore, reading documentation carefully is important; sometimes it's also good to just play around and find out small details like this.

A typical workflow is, we load the operands from memory to registers first, and use `ADD` or `SUB` to perform the calculation. ALU will send the result back to the destination register, so we'll have to store the result back from the register to memory.

Example 2.3

Assume a long integer `var` is stored at address `ptr_var` in memory. The value of `ptr_var` has been loaded into register `X9`. Write a sequence of ARMv8 assembly to add 1 to `var`. This is similar to translating the following C code into assembly:

```
1 long int  var;
2 long int* ptr_var = &var;
3 *ptr_var ++;
```

Solution: To perform any type of calculation, the operands have to be inside the registers. Since we want to perform addition on `var` which is currently inside the memory, we need to first load it into a register, say `X10`: `LDUR X10,[X9]`. For the addition, one operand now is already `X10`, but the other operand is number 1, which is an immediate, so we can use `ADD` with an immediate operand.

Now that two of the operands of the addition are ready, we also need to decide which register to store the result of addition. Assume we want to save it to `X12`: `ADD X12,X10,1`.

After this instruction, we are not done yet, because the result

currently resides in a register, but what we want is to update the variable which resides in memory. Therefore, we need to move the result back by using `STUR` instruction. Notice that at this moment `X9` still stores the address of `var`, so if we write back there, it'll overwrite the old value of `var`: `STUR X12,[X9]`.

In sum, the sequence looks like this:

```
LDUR   X10, [X9]
ADD    X12, X10, 1
STUR   X12, [X9]
```

Remarks: You can certainly set the destination register for `ADD` to `X10`, which will overwrite the old value of `X10`, basically like `x10 = x10 + 1`. However, that `X10` was updated with new value **does not** mean the value inside the memory `var` will also be updated. Therefore, you have to `STUR` the result back manually.

We don't usually use multiplication and division in our course, but we will show the instructions here for reference:

- `MUL Xd,Xn,Xm`: `Xd = Xn * Xm;`
- `SDIV Xd,Xn,Xm`: signed *integer* division, `Xd = Xn / Xm;`
- `UDIV Xd,Xn,Xm`: unsigned *integer* division, `Xd = Xn / Xm`.

2.4.2 Logic Operations

Another type of calculation that the ALU can perform is logic operations, which is basically bit-wise operations. The instructions we can use are:

- `AND Xd,Xn,Xm`: bit-wise `and` on registers `Xn` and `Xm` and store the result in `Xd`;
- `ORR Xd,Xn,Xm`: bit-wise `or` on registers `Xn` and `Xm` and store the result in `Xd`;
- `EOR Xd,Xn,Xm`: bit-wise `exclusive-or` on registers `Xn` and `Xm` and store the result in `Xd`.

All these instructions can also be used for `W`-registers, but all three operands need to match the size.

In addition to those mentioned, we also sometimes want to shift the bits in a register, left or right. Recall that we have two types of shifting—logical and arithmetic.

Correspondingly, our instructions have these versions as well:

- `ASR Xd, Xn, Xm`: arithmetic shift right `Xd = Xn >> Xm;`
- `LSL Xd, Xn, Xm`: logical shift left `Xd = Xn << Xm;`
- `LSR Xd, Xn, Xm`: logical shift right `Xd = Xn >> Xm`.

As discussed in Section 1.1.5.2, if we want to multiply a number by 2^n, the fastest way is to shift left using `LSL`. Also, register `Xm` in the three instructions above can also be replaced by immediate numbers.

Quick Check 2.3

Congratulations! You have learned enough assembly instructions to write many programs! See if you can write sequences of assembly instructions for the following problem.

(Fall 21 quiz) Write the corresponding ARMv8 assembly code for the following C statement. Assume that the variable `f` is in register `X20`, and the base address of array `A` is in register `X21`. `A` is an array of integers. `A[0] = f + A[5];`

2.5 Flow Control

One huge difference between the high-level languages you have learned and assembly is that the former is highly *structured*, whereas the latter is just a sequence of instructions. Therefore, repeat after me → "I will forget about everything I learned about program structures, such as `for`-loop, `if-else` statements, and so on."

The assembly program simply executes one instruction after another **sequentially**, unless we explicitly to instruct the program to jump to a specific instruction. Assembly programs don't have "memory" either: you cannot expect the program will magically jump back to somewhere without your instruction.

Once we're clear on this, let's talk about program counter first.

2.5.1 Program Counter

The name of **program counter (PC)** might be a little bit confusing; its job is not exactly to "count" the program. PC is a 64-bit special register inside CPU, and no we cannot modify the data inside it.

You probably already know this: when we're executing our program, the assembly instructions are also stored in RAM. Because everything stored in RAM has an address, each instruction in our assembly programs also has its own address.

For us humans, when we read/write an assembly program, we know after one instruction we just go to the next one, but how does machine know?

Here's what happens: the CPU will store **the address of its next instruction to be executed** into PC. After the current instruction has finished, the CPU will simply use the address inside PC to go to RAM and retrieve the next instruction. [3]

3: Disclaimer: this is an extremely simplified example. Later in Chapter 3, you'll see it's more complicated. However, to not get confused, it's entirely ok to think about it this way now.

Example 2.4

Consider the following assembly sequence,

```
1 ADD   X9,  X10, X11   /* 0xFFFF0040 */
2 SUB   X11, X12, X9    /* 0xFFFF0044 */
3 STUR  X9, [X11, 0]    /* 0xFFFF0048 */
4 LDUR  X9, [X12, 8]    /* 0xFFFF004C */
```

When executing this program, all instructions will be stored in to memory contiguously. Because each instruction takes four bytes, the addresses of these instructions are multiples of four.

Before executing the first instruction `ADD`, the PC has a value of `0xFFFF0040`. When the `ADD` instruction starts running, PC is automatically incremented by four bytes, which makes it `0xFFFF0044`. When `ADD` finishes, CPU looks at the address inside PC which is `0xFFFF0044`, and goes to that memory location to fetch the next instruction, which is `SUB`.

When `SUB` starts running, PC is again incremented by four bytes, making it `0xFFFF0048` that points to the next instruction. As you see, the only thing that the CPU looks at is PC; it'll simply fetch whatever pointed by PC.

We cannot manually give PC a value like moving some value to a register, because PC determines the program flow. If we can send arbitrary 32 bits to PC (that cannot be decoded as a valid instruction), it'll make interrupt the normal program flow, and maybe even crush the system. But sometimes we just want to skip some instructions, which surely needs us to "modify" PC. This can be achieved by using special instructions.

2.5.2 Branching

Assembly programs are sequentially executed unless specified. This is entirely done by instructions that can modify PC.

2.5.2.1 Unconditional Branching

The simplest way is to use `B` instruction (stands for **b**ranching). The syntax is `B target` where `target` is the target instruction's memory address. The problem is we don't really know the memory address of each instructions when they're running. Also it's more tiring for us to "count" or "calculate" the address. Therefore, we can use a label (if you forgot about labels, go back to Section 2.1).

Example 2.5

Consider the following assembly sequence,

```
1       ADD   X9,  X10, X11   /* 0xFFFF0040 */
2       B     L1              /* 0xFFFF0044 */
3       SUB   X11, X12, X9    /* 0xFFFF0048 */
4  L1:  STUR  X9, [X11, 0]    /* 0xFFFF004C */
5       LDUR  X9, [X12, 8]    /* 0xFFFF0050 */
```

where we added a label `L1:` for the `STUR` instruction. On line 2, `B L1` means we want to execute the instruction pointed by `L1` next instead of its following instruction `SUB`.

When executing instruction `B L1`, because the instruction says "now you need to go to `L1`", PC will be changed to the address of `L1` which is `0xFFFF004C`, instead of the address of `SUB` which is `0xFFFF0048`.

Therefore, after `B` instruction is done, the CPU fetches the target instruction at `L1` (the `STUR`), and start running from there. See Figure 2.5 for an illustration.

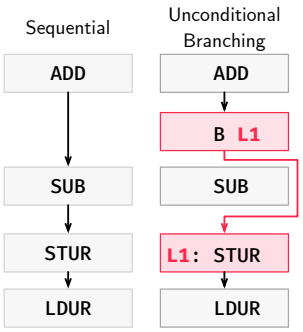


Figure 2.5: Program flow of the example. Instruction `B` modifies PC based on its target, which makes the program skip some instructions.

In addition to `B`, if the address has already been stored in a register, say `X20`, we can use another instruction `BR` (**b**ranch from **r**egister): `BR X20`.

2.5.2.2 Conditional Branching

The unconditional branch could be useful sometimes, but if we want to write something like `if-else`, we'll have to use conditional branch.

Consider the following C code snippet:

```
1 long a;
2 if (a == 0) a ++;
3 a --;
```

There are two possible paths for the program: one goes through `if` block, while the other one doesn't. If we draw a flowchart, it'll be like the middle diagram in Figure 2.6.

Given this flowchart, we just need to translate each part into their corresponding assembly instructions. Assume variable `a` has an address stored in `X0`, and we want to load this variable into `X9`. The `if` block is simple: we just need to add 1 to register `X9`. For the last statement, it's the destination of branching (if `a!=0`), so we need to add a label in front of it, say `L1`.

Now we only have one problem left: how to translate the condition, *i.e.*, the `if` statement, and this is where we'll introduce the first set of conditional branching instructions. Currently, variable `a` is in `X9`, so we just need to check if `X9` is zero or not; if it is zero, we keep moving on to the

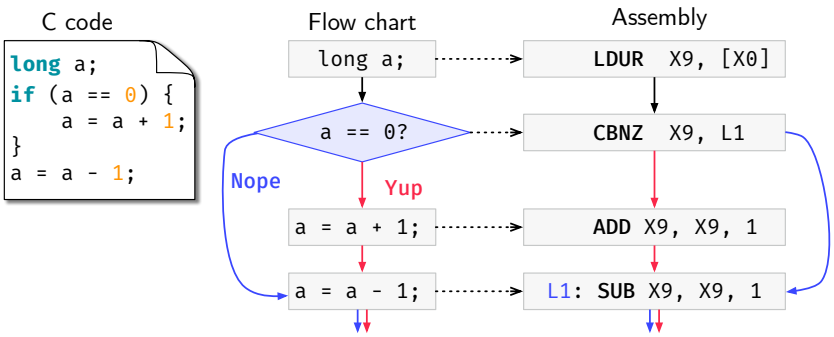


Figure 2.6: CBNZ will check the register value; if it's not zero, it'll branch to the instruction tagged as L1.

next instruction (ADD); otherwise we "jump" or branch to the instruction labeled as L1. The instruction to do this is CBNZ (conditionally **b**ranch if **n**ot **z**ero):

```
1 CBNZ   Xt, Label
```

This instruction will check the value inside Xt. If it's not zero, it'll move the address of the instruction tagged with Label to PC, and thus branches to that instruction. If it is zero, it'll just move to the very next instruction.

Another related instruction is CBZ, which you can probably guess that it'll jump to the destination if the register it's checking is zero.

Now that we've introduced the branching instructions (at least a small portion of it), there are some common beginner mistakes that we need to be careful of.

Let's re-write the C code in Figure 2.6 using CBZ instruction. Following our steps before, the instruction ADD is the target of CBZ, so it's natural to give it a label, say L2. So someone came up with the following sequence:

```
1       LDUR  X9, [X0]    /* long a; */
2       CBZ   X9, L2      /* if (a == 0) goto L2; */
3  L2:  ADD   X9, X9, 1   /* a = a + 1; */
4  L1:  SUB   X9, X9, 1   /* a = a - 1; */
```

Let's walk through this sequence. If X9 is indeed zero, the flow of instruction would be LDUR → CBZ → ADD → SUB, which is correct. If X9 is not zero, the flow is also LDUR → CBZ → ADD → SUB, which is not what we want obviously. The problem is we didn't skip the ADD instruction, so a quick and simple fix: after CBZ, add a CBNZ and jump to L1:

```
1       LDUR  X9, [X0]    /* long a; */
2       CBZ   X9, L2      /* if (a == 0) goto L2; */
3       CBNZ  X9, L1      /* if (a != 0) goto L1; */
4  L2:  ADD   X9, X9, 1   /* a = a + 1; */
5  L1:  SUB   X9, X9, 1   /* a = a - 1; */
```

Now the flow when a != 0 is correct: LDUR → CBZ (failed) → CBNZ → SUB.

Example 2.6

Translate the following C code into assembly:

```c
long a;
if (a == 0) a = a + 2;
else a = a - 2;
a = a * 4;
```

As usual, we first draw the flowchart of the C code, as in Figure 2.7. Notice this time because the statement `a = a - 2` is in the `else` block, after we finish the `if` block, we'll have to skip the statements in the `else` block.

With the flowchart, we can clearly see there are two branches, and thus two destinations: one is `a = a - 2` for the `else` case, and the other is `a = a * 2` when `if` case has finished. Therefore, when translating to assembly, we will add two labels for each of the destinations, `L1` and `L2`.

We use `CBNZ` to test the `if` statement like what we did before. After `ADD` instruction, we reached the end of `if` block, and need to skip `else` block and jump to `L2`. This time, this branching is unconditional — it doesn't depend on a specific value or register, and therefore we use unconditional branch `B L2` to force the program dodge `else` case.

Common mistakes: as shown in the rightmost diagram in Figure 2.7, a common mistake is that we forgot to add unconditional branch at the end of the `if` block. Remember, assembly is very innocent — if you don't specify it to branch/jump to somewhere, it'll just execute the very next instruction. Without `B`, you can see the red flow passes through all instructions, and thus the wrong result. No, assembly does not recognize `if-else`, and does not remember which instruction is the end of what block. All the control is in your hand, not the machine's.

Take-away: unlike high-level languages which usually have `{}` to determine the flow of the program, assembly will always execute the instructions by order unless we explicitly specify the flow.

Surely comparing a register with zero is helpful for creating branches, but that's not enough for more complicated cases, such as `if (a < b)` and so on. Let's start with a simple example.

Consider the following C code:

```c
if (a >= b) a = a + b;
else a = a - b;
```

Assume variable `a` and `b` are in registers `X9` and `X10`. To implement the `if`

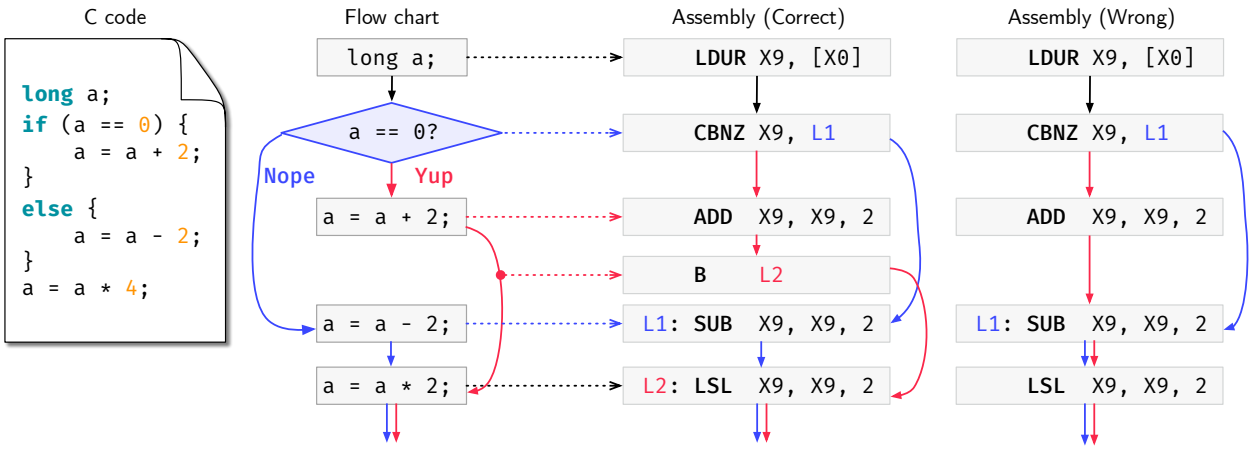


Figure 2.7: Without unconditional branch `B`, the program flow would be wrong when variable `a` is zero.

condition in assembly, we need two steps: (1) compare the two numbers, and (2) jump to destination based on some criteria. The instruction for step (1) is `CMP`:

- `CMP Wn, Wm`: subtract `Wm` from `Wn`;
- `CMP Xn, Xm`: subtract `Xm` from `Xn`.

So we can write `CMP X9, X10` for this step. What this instruction does is to subtract `X10` from `X9`, and compare the result with zero. Thus it's equivalent to checking `if (a - b >= 0)`.

Next, we use a condition branch instruction `B.LT` (**b**ranch if **l**ess **t**han). This could mean two equivalent things: branch if `a` is smaller than `b`, or branch if `a - b` is smaller than zero.

In sum, the assembly sequence looks like this:

```
1       CMP   X9, X10      // Check X9 - X10 (a - b)
2       B.LT  Else         // If a - b < 0, branch to Else
3       ADD   X9, X9, X10  // Otherwise (a >= b), a = a + b
4       B     Done         // Finish if-block
5  Else: SUB  X9, X9, X10  // a = a - b
6  Done: ...               // Out of if-else block
```

Remember, `CMP` command simply compares the two operands; it doesn't make any decisions; it is us who use `B.LT` instruction to change the flow of the program.

You can probably guess that in addition to `B.LT`, we also have other instructions for "greater than", "greater than of equal", *etc.* Before looking at those instructions, however, We need to take a bit detour to look at condition codes. Have you wondered, how does the machine know the result of `CMP` is negative? And how does `B.LT` instruction know the comparison result of previous `CMP` instruction? The magic behind these is condition codes.

2.5.2.3 Condition Codes

Inside CPU, there's a special register called **CPSR**, or **current program status register**, which stores 32 bit. Different than the registers we have seen, these 32 bits are used *individually*, and typically directly set by the processor (not us!). We only care about the highest four bits, which correspond to the four condition codes respectively:

31	30	29	28	0 – 27
`N`	`Z`	`C`	`V`	Other flags
Negative	Zero	Carry	Overflow	

When a condition code has a value of 1, we say it's **set**; otherwise it's **clear**.

The four condition codes indicate the following situations:

- `N`: negative (for signed number), a copy of the MSB of the calculated result;
- `Z`: set if the result is zero; otherwise clear;
- `C`: set if there's a carry for unsigned numbers;
- `O`: set if there's an overflow for signed numbers.

Facts about condition code you need to remember:

1. Condition codes are changed by instructions after the instruction has been executed;
2. Not all instructions change condition codes;
3. Different instructions change different condition codes;
4. The condition codes that are not changed by an instruction will **not** be cleared automatically; it stays until some other instructions changed it;
5. The condition codes are changed individually.

In sum, condition codes reflect the status change of instructions just being executed. Notice that, however, not all instructions are designed to change condition codes. So far the only instruction we learned that changes the codes is `CMP`.

Arithmetic/logic instructions such as `ADD` and `ORR` do not modify condition codes. If we want them set condition codes, we need to use the following instructions which explicitly set condition codes: `ADDS`, `SUBS`, and `ANDS`.[4]

4: None of the logical instructions including shifting modifies condition codes, except `ANDS`.

2.5.2.4 More Conditional Branch Instructions

So now the magic of `CMP` has been revealed: `CMP` instruction simply does the subtraction and changed the condition codes correspondingly. Then the set of instructions in Table 2.1 will take a look at the condition codes and check if the status of these codes satisfy the condition to branch.

Table 2.1: Conditional branch instructions `B.cond` and condition code checking.

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
$=$	`B.EQ`	`Z == 1`	`B.EQ`	`Z == 1`
$\neq$	`B.NE`	`Z == 0`	`B.NE`	`Z == 0`
$<$	`B.LT`	`N != V`	`B.LO`	`C == 0`
$\leq$	`B.LE`	`~(Z == 0 && N == V)`	`B.LS`	`~(Z == 0 && C == 1)`
$>$	`B.GT`	`(Z == 0 && N == V)`	`B.HI`	`(Z == 0 && C == 1)`
$\geq$	`B.GE`	`N == V`	`B.HS`	`C == 1`

One possible question is, for example, when comparing $\leq$, we have signed numbers and unsigned numbers. How does the computer know if a data is signed or unsigned? The answer is, it doesn't know, or it doesn't need to know. Remember we emphasized this from very beginning: everything inside memory, either it's code or data, or no matter what kind of data, is just binary sequence. Surely when we write a C program, we can specify something like: `int a = -1; unsigned int b = 4294967296;` which makes variable `a` a signed number and `b` an unsigned number. However, if you look at how they were stored in the machine, both of them are exactly the same: `0xFFFFFFFF`. The machine has no idea which one is what.

Then who decide which number is signed or unsigned? Of course it's us. If you treat the numbers you want to compare as unsigned, then just use for example `B.LS`; if you want to take them as signed, then use `B.LE`. From Table 2.1, you see both `B.LE` and `B.LS` are comparing $\leq$, but the only difference is that these two instructions check different condition codes. Instructions such as `CMP` will modify the condition codes regardless, then it's up to us how we want to treat our numbers and thus which instruction to choose.

Example 2.7

Note for this example, we just pretend X-registers are 8-bit wide for convenience; they are of course 64 bits in fact. Assume currently in register `X9` we have data `1111 1011` whereas `X10` we have `0000 0101`. Consider the following assembly code:

```
1       SUBS   X11, X9, X10
2       B.LT   L1
3       B      L2
4       ...
5   L1: ...
6       ...
7   L2: ...
```

After running instruction `SUBS X11,X9,X10`, we have the following values in the registers:

Register	Data	Signed	Unsigned
X9	1111 1011	−5	251
X10	0000 0101	5	5
X11	1111 0110	−10	246
N = 1, Z = 0, C = 1, V = 0			

Because X11 is not zero and there's no overflow, both Z and V are clear. The MSB of X11 is 1, and it'll be copied to N, meaning if it *was treated as a signed number*, it'll be negative. Lastly, C is also set because a carry happened for the 3rd bit.

Now, on line 2, we used B.LT instruction, meaning we want to treat X9 and X10 as signed numbers. B.LT then will examine condition codes N and V. In this example, N != V, so the program will successfully jump to label L1.

If we, however, use B.LO on line 2 instead, the instruction ignores N and V, and only checks if C is equal to zero. In this example, C is 1, so the instruction will not jump to L1, and will keep executing line 3 instead, which brings us to label L2.

From this example, you can see when running instructions that set condition codes, the machine has no idea if the operands are signed or unsigned at all. It'll do its job and set the condition codes based merely on bit operations. It is *our job* to decide to treat them as signed or unsigned, and use instructions accordingly.

Quick Check 2.4

(Fall 21 Quiz) Write an ARMv8 assembly code that sets x to y + 10, if y is equal to 1, otherwise sets x to y/8. Assume that x and y are in X19 and X20, and are treated as signed integers. Note: you don't need multiplication or division instructions to finish this question at all.

2.5.3 Loops

You probably already figured out that loops are basically just branching, but backward instead of forward. Let's look at the following C code:

```
1 long i = 0;
2 while (i < 10) i = i + 1;
```

Using a C keyword `goto`, we can rewrite the code as follows:

```
1 long i = 0;
2 Begin:
```

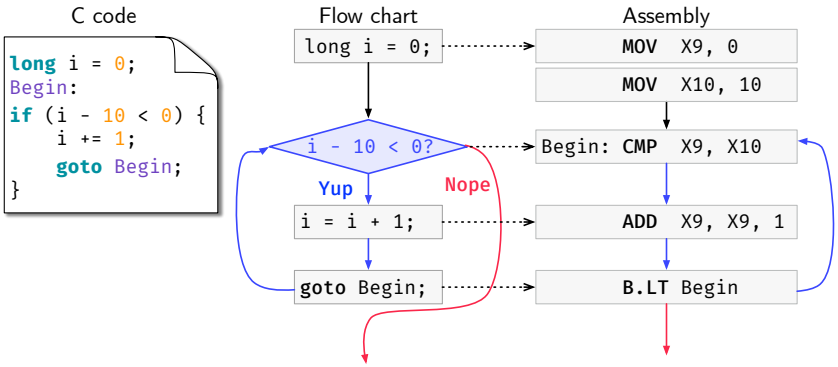


Figure 2.8: A loop is simply a backward branching.

```
3     if (i - 10 < 0) {
4         i = i + 1;
5         goto Begin;
6     }
```

This way, we convert the loop into a conditional branch we're familiar with, and therefore the translation to assembly is straightward. See Figure 2.8 for the assembly code.

However, there are some minor stuff in Figure 2.8 that's worth mentioning. This time, we didn't use `LDUR` in the beginning. This is fine, and just assume we're going to use `X9` for variable `i`. Before the comparison, we also set `X10` to 10, because `CMP` instruction does not accept immediate operands. [5] The `ADD` instruction does not change condition codes, so we can use `B.LT` after `ADD`. However, if you use `ADDS`, the logic is wrong and the program will potentially fail. Lastly, we use `B.LT` because when we declare `long` integer type in C, it's signed number by default.

5: You can certainly use `SUBS` instead, *e.g.*, `SUBS X0, X9, X10`.

2.5.4 Arrays

Dealing arrays almost always needs loops. Let's start with the simplest case, *i.e.*, character arrays (or strings), since each character takes only one byte.

2.5.4.1 Strings

Say we have five characters, `"qwert"`, stored in RAM. We want to add 2 to each of the characters, so that it becomes `"sygtv"`. The C code for this task is not difficult:

```
1 char str[] = {'q','w','e','r','t'};
2 for (int i = 0; i < 5; i ++) {
3     *(str + i) = *(str + i) + 2;
4 }
```

Now let's write the assembly for this task!

1. *Analysis:*
   From our C knowledge, we already know `str` is a pointer, and its

value is the address of the first byte of the array. Therefore, `str` is the **base address** of the array. Because each character takes one byte, the index `i` of each character is also the **offset** of that character from the base address. Thus, `str + i` is the address of the `i`-th character.

2. *Panning:*
Now, let's assume the address of `str` is already loaded into `X9`. Other temporary data that need to be stored are `i` and constant array length 5. So in summary, we have register usage shown in the table on the side.

Register	Data
`X9`	base address of `str`
`X10`	length 5
`X11`	index `i`
`X12`	`i`-th element `*(str+i)`

The pseudocode or assembly algorithm should look like this:

Algorithm 1: Pseudocode for iterating strings

```
1 Set X10 to the array length 5 → MOV X10,5 ;
2 Set X11 to zero for the counter i → MOV X11,0 ;
3 while X11 < X10 do
4     Load *(str+i) into register X12 →
        LDURB X12,[X9,X11] ;
5     Add 2 to *(str+i), and put it back to X12 →
        ADD X12,X12,2 ;
6     Store X12 back to memory → STURB X12,[X9,X11]
      Update X11 (i++) → ADD X11,X11,1 ;
```

3. *Assembly:*
All the direct translations of assembly have been added to the pseudocode for clarity. Now the only thing left is to complete the loop. Notice the first instruction of the loop is `LDURB`. Therefore, we can add a label to that instruction, say `Begin`. At the end of the loop, we need to check if `X11` is already equal to 5. The complete code will look like this:

```
        MOV    X10, 5
        MOV    X11, 0
Begin:  LDURB  X12, [X9, X11]
        ADD    X12,  X12, 2
        STURB  X12, [X9, X11]
        ADD    X11,  X11, 1
        CMP    X11,  X10
        B.LO   Begin
```

2.5.4.2 Larger Types

Strings are relatively easy, because each element of the array takes only one byte, and the index of the array can thus be conveniently used as offset from the base address. When a data type takes more than one byte, it needs to be planned more carefully.

Now we'll modify the example in the previous section with a `long int`

type array, where each element takes eight bytes. The task is still to add 2 to every element in the array:

```
long arr[] = {30, -23, 100, 99, 0};
for (int i = 0; i < 5; i ++) {
    *(arr + i) = *(arr + i) + 2;
}
```

Let's follow the three steps again — analysis, planning, and assembly — to translate this C code into assembly.

1. *Analysis:*
   From previous example, we've been fairly familiar with arrays, so no doubt `arr` is the base address of the array. What's different here is, the index `i` *cannot* be used as offset anymore. When it's a string, moving the index by one also means moving/offsetting to the next one byte, and therefore the index can be conveniently used as offset as well.

   In this example, however, each element takes eight bytes, meaning when we move on to the next element (say `arr[i+1]`), we should skip eight bytes of the current element (`arr[i]`) so that we can land on the first byte of the next element. If we still use the `i` as offset, we are actually only moving to the next *byte*, instead of next *element*.

2. *Panning:*
   Based on the analysis above, we are clear that before using `LDUR` or `STUR`, we have to calculate the offset correctly. Given an index `i`, if each element takes eight bytes, the correct offset should be `i * 8`. Thus, we have the planning for registers as shown on the side.

   The pseudocode is very similar to the previous example, except that it needs an additional step to calculate the offsets correctly for each element. Also notice this time because we're using full eight bytes, we use `LDUR` and `STUR` instructions.

Register	Data
`X9`	base address of `str`
`X10`	length 5
`X11`	index `i`
`X12`	`i`-th element `*(str+i)`
`X13`	offset `i*8`

Algorithm 2: Pseudocode for iterating long integers

```
Set X10 to the array length 5 → MOV X10,5 ;
Set X11 to zero for the counter i → MOV X11,0 ;
while X11 < X10 do
    Calculate offset → LSL X13,X11,3 ;
    Load *(arr+i) into register X12 →
      LDUR X12,[X9,X13] ;
    Add 2 to *(arr+i), and put it back to X12 →
      ADD X12,X12,2 ;
    Store X12 back to memory → STUR X12,[X9,X13] ;
    Update X11 (i++) → ADD X11,X11,1 ;
```

3. *Assembly:*
   With the analysis and planning, translating to assembly is very straightforward:

```
1         MOV    X10, 5
2         MOV    X11, 0
3  Begin: LSL    X13,  X11, 3
4         LDUR   X12, [X9,  X13]
5         ADD    X12,  X12, 2
6         STUR   X12, [X9,  X13]
7         ADD    X11,  X11, 1
8         CMP    X11,  X10
9         B.LO   Begin
```

Quick Check 2.5

(Fall 21 Final) Assume there's an array:
`long arr[] = {12, 34, 56, 78}`
Write an assembly to store the numbers in the reverse order in another array `arr2`. Suppose `arr` is already in `X9`, and `arr2` in `X10`. You have to use a loop to complete the task.

2.6 Procedures

In this section we're going to learn how to write procedures in assembly.

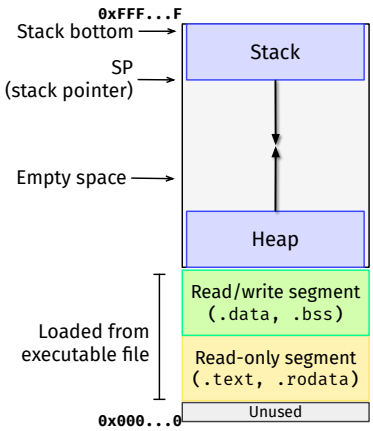


Figure 2.9: Visualization of a virtual memory space for a program. Our assembly code and global variables will be loaded to this space straight out of the executable file. During run time, the heap and stack are growing towards each other as our program calls a procedure, or allocates space dynamically. For stack, the bottom is actually at the high address, while the top is at the low address, so you can take it as a upside-down stack.

Before talking about procedures, let's review the memory address space first. Whenever we start a program, we assume that the program takes the entire **virtual memory space** from address `0x00..0` to `0xFF..F` in the memory. At the bottom of this space, we have `.text` segment that stores our assembly code and read only data (*e.g.*, string literals). Going up, we have `.data` segment that stores global variables. After `.data` segment we also have `.bss` used for uninitilized data. These parts are basically predetermined during **compilation time**.

The rest of the space will be used during **run time**, meaning they're used for storing and managing data that only occurs when the program is actually executing, such as local variables. The start of the area is called **heap**, which is used for dynamic allocations during run time. The end of the area is called **stack**, used for procedure calls. Since they occupy the two ends of the empty space of the memory, they grow towards each other, or towards the center. Figure 2.9 shows a bit more detailed visualization.

Since we're going to write procedures, we'll natually focus on the stack area, and see how calling or returning from a procedure manages the stack.

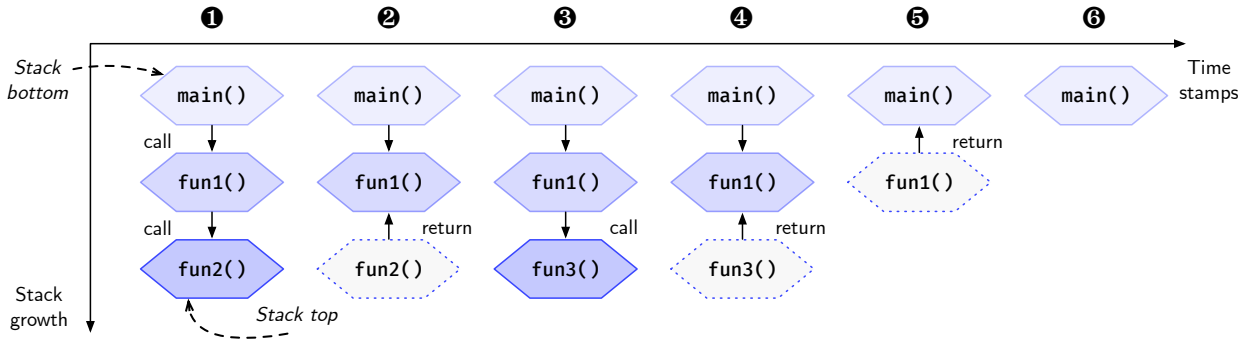


Figure 2.10: If we treat function/procedure calls as stacking some "blocks", the process of procedure calls looks really like pushing blocks to the stack. At point (2), `fun2()` returned to `fun1()`, and therefore its block is removed from the stack top.

2.6.1 Runtime Stack

Through Data Structures you have learned that stack is a First-In-Last-Out (FILO) structure, and the main operations for stacks are push and pop. **Push** is to add something to the top of the stack, whereas **pop** is to take the top element of the stack out.

Stack is also the structure for procedure calls. Intuitively, there are a lot of similarities of behaviors between stacks and procedures. Let's look at the following example.

Assume we have a C code where we have three functions called `funx()` ($x = 1, 2, 3$)). `main()` calls `fun1()` and `fun3()`, and `fun1()` calls `fun2()`:

```c
int main() {
    fun1();
    ...
    fun3();
    return 0;
}
```

```c
int fun1() {
    ...
    fun2();
    ...
    return 0;
}
```

Assume every time we call a procedure, we put a "block" at the top of a stack, and every time we return from a procedure, we take that block out. Formally, `fun1()` is the **caller** of `fun2()` and `fun3()`, and these two functions are the **callee** of `fun1()`. Figure 2.10 shows us a time line from when `fun2()` was called to `fun1()` returned to `main()`.

Now think the "blocks" as boxes where each procedure stores its own local variables. These boxes, called **procedure frames**, or simply **frames**, are exactly how the system manages stack space for procedure calls as in Figure 2.9.

One question is then, are those frames created automatically when we call a procedure? The answer is no. Frames are nothing but a designated area on stack for procedure management; it is basically in our head. When we call a procedure or return from a procedure, we have to manually allocate/de-allocate the frame area on stack. We don't need to do this in high-level languages but in assembly we have to manage these areas.

2.6.2 Procedure Call Conventions

Simply put, procedures in assembly are just branches with return. Let's look at the following example.

```
1 Proc:    MOV  X0, 0
2          MOV  X1, 1
3          B    Ret_Pt
4 _start:  B    Proc
5 Ret_Pt:  MOV  X1, 1
6          ...
```

In the code listing above, line 1 to line 3 can be treated as a procedure called `Proc`. You can see it's nothing special but a branch.

We start running our program from label `_start`, where we branch to `Proc`. `MOV X0,0` is the first instruction of the procedure `Proc`. This is similar to "call" the procedure. Then at line 3, we unconditionally branch to `Ret_Pt`, which is the next instruction of `B Proc`, and move on from there.

Question: what if we forgot to write line 3, and didn't branch back to `Ret_Pt`? What will happen?

This simple code is a very simple procedure call (though not complete), but it shows us the first essential element of procedure call: **return address**.

2.6.2.1 Return Address

What's special about procedure is, after the callee finishes, the program will return back to where it was called in the caller, and keep executing from there. Since in assembly every instruction has an address, we need to know where we should go back to from the procedure. In the example above, you can see we should return to the next instruction from the branch instruction `B Proc`, and therefore `Ret_Pt` marks the return address, and we branch back to `Ret_Pt` from `Proc`.

Using a label to mark the return address is an ok option, but not the best. If you have a billion of functions, you'd have to create a billion of labels for their return addresses. Too much labels only make programs hard to follow. Plus there are also portability issues.

A much better option is to use `BL` instruction (***b***ranch and ***l***ink). As the name suggests, it does two things: branch to a label, and link the return address. Using `BL` instruction, we modify the example code:

```
1 Proc:    MOV  X0, 0
2          MOV  X1, 1
3          RET
4 _start:  BL   Proc
5          MOV  X1, 1
6          ...
```

What `BL` instruction does is first push the address of its *next* instruction (*i.e.*, the return address) into register `X30`, and then branch to `Proc`. Notice by convention, `X30` (can also be referred as `LR`) is called **link register**, whose purpose is exactly to store return address. Therefore, do not use `X30` for other purposes.

The second change we did is on line 3 where we now use `RET` instruction, which obviously stands for "return". Because `X30` is used by default for return address, `RET` instruction will simply copy `X30` to PC, so that next instruction executed is the one at the return address.

2.6.2.2 Passing Arguments and Return Values

The second essential element of a procedure is passing arguments/return values. In the code example above, we didn't pass any arguments, but in practice it is quite common.

The quickest way to pass arguments is through registers, because it doesn't involve reading/writing memory. Again, by convention, registers `X0` to `X7` are used to pass arguments. Of course there's no hard requirement that you have to pass arguments through those eight registers, but portability/-compatibility becomes the issue again, so we'll just ask you only use `X0` to `X7` to pass procedure arguments.

The above applies to passing arguments but also return values.

Example 2.8

Write an assembly program that translates the following C code:

```
1 long addition(long a, long b) { return a + b; }
2 int main() {
3     long a = 10;
4     long b = 20;
5     long c = add(a, b);
6 }
```

Assume addresses of variables `a`, `b`, and `c` are stored in registers `X9`, `X10`, and `X11`, respectively.

Solution: First step is still to write `LDUR` instructions for variables `a` and `b`. We don't need to `LDUR` variable `c` because its value is not stored in the memory. We'll use `_start` as the entrance of the main procedure:

```
1 _start:  LDUR  X0, [X9]    // Load a to X0
2          LDUR  X1, [X10]   // Load b to X1
```

Before we branch to the procedure, we need to move arguments we want to pass to registers `X0` – `X7`. In this example, luckily, when we load the two variables they are already in `X0` and `X1`, so we

don't need to move them anymore. The only thing we need is to branch to the procedure. Let's call the procedure `Add`, then the next instruction would be `BL Addition`.

The instruction following this `BL` is where we return from the caller; that's also where we grab the return value from the callee. We'll store this into variable `c` in memory, so a natural step is to use `STUR`: `STUR X0,[X11]`. In summary, the main procedure looks like this:

```
_start:LDUR  X0, [X9]    // Load a to X0
       LDUR  X1, [X10]   // Load b to X1
       BL    Addition    // Call add() function
       STUR  X0, [X11]   // store return value to c
```

The procedure `Addition` should also be easy. Since we followed the standard by passing two arguments through `X0` and `X1`, the procedure can just use them directly. Also because the return value is stored in `X0`, we can just let the destination of `ADD` instruction be `X0`. Thus, we have:

```
Addition:  ADD  X0, X0, X1  // a = a + b;
           RET              // Return to the caller
```

Note that it doesn't matter if you write main procedure first or `Addition` first; their placement in your source code doesn't matter. In the above code we did not add appropriate terminating statements in the main procedure, so it could cause a problem if you put `Addition` below `STUR`. Please refer to section B.1.1 in the appendix for more details on how to terminate a program.

2.6.2.3 Creating Frames

In the previous examples we didn't create any frames for procedure calls, because the registers are enough to perform what we want. More than often, though, registers are never enough to store all the data needed for procedures. Thus, the purpose of creating frames is to store procedure arguments and local variables.

Remember, however, that frames are merely a *way* of managing procedure calls; the computer system has no idea what a frame is, or where the frame boundaries are at. Which means it is our job to know where the frame is. So how?

Frames are just an area on stack in the memory, and again, everything in the memory has an address. If, for a procedure call, we know where the first and last bytes of its frame are at, we are able to draw a boundary and claim something like, the area between `xyz` and `abc` is the frame area for this procedure call.

Back to Figure 2.9, we see that there's a **stack pointer** that points to the top of the stack, and this is exactly the "boundary" we need. Stack pointer stores the address of the current lowest byte of the stack, and is inside register `X29` (also referred as `SP`). Note that this register can certainly be used for other purposes and the system doesn't prevent you from using it, but as a good practice and the sane of your code (and your mental health), do not use `X29` other than storing stack top address.

The other "boundary" for procedure frames is **frame pointer**, stored in `X28` or referred as `FP`. Thus, the bytes between `FP` and `SP` form the area we visualize as procedure frame.

Are `SP` and `FP` automatically changed when we `BL` to a procedure? Sorry nope! We have to manually change `FP` and `SP` on our own. In fact we can only need to change `SP` since it points to the stack top and thus more important; changing `FP` is optional, as long as you remember the size (*i.e.*, how many bytes) of the procedure frames.

☢ Caution!

Based on ARMv8 documentation, the size of procedure frames have to be multiples of 16 bytes. See Section 2.6.4.

❝ Example 2.9

Let's use this example to show how procedure frames work in detail. We'll start from a C code again:

```c
void proc(long length) { long arr[length] = {-1}; }

long fun(long a, long b) { return a + b; }

int main() {
    proc(20);
    long x = fun(2,3);
}
```

The main procedure and `fun()` should be easy to produce at this point:

```
fun:     ADD   X0, X0, X1 // a + b
         RET

_start:  MOV   X0, 20     // Pass 20 as parameter
         BL    proc       // Call proc(20)
         MOV   X0, 2      // Parameter 1 <- 2
         MOV   X1, 3      // Parameter 2 <- 3
         BL    fun        // Call fun(2,3)
         STUR  X0, [X9]   // Store long x,
                          // assume &x is in X9
```

In `proc()`, we created an array of `length`, and initialized every element to negative one. This array is a local variable, meaning we need to store it on stack. More specifically, because this array is local in `proc()`, we have to store it in the procedure frame on stack. So let's create a frame for it now!

First thing: how many bytes we need for the frame? Each element is `long` type which takes eight bytes. Since there are `length` elements, total size of the array is $8 \times$ `length`. There's no other local variables, so we can just allocate $8 \times$ `length` bytes on the stack.

Currently `SP` is pointing to the stack top. If we want to add, or *push*, a frame to a stack, we need to **subtract** `SP` because remember the stack grow towards **low address**. At the end, before we return, we need to "de-allocate" the frame, so we add the corresponding number of bytes back. Thus, we have the following framework:

```
1  proc:  LSL   X9, X0, 3   // length * 8
2         SUB   SP, SP, X1  // Allocating frame:
3                           // SP = SP - length*8
4
5         /* --- Main procedure body --- */
6
7         ADD   SP, SP, X9  // De-allocating frame:
8                           // SP = SP + length*8
9         RET
```

The code above does need us to be careful: at the end we add `X9` back, which is supposed to be the size of the frame (calculated in line 1). This means inside the function we cannot change the value of `X9`; or we have to store the value of `X9` somewhere so we can retrieve it later.

The procedure body is just a loop over `length` elements. Notice every element takes eight bytes, so the structure would be like the example in Section 2.5.4.2. To put it together, we have the following for `proc`:

```
1  proc: LSL   X9, X0, 3   // length * 8
2        SUB   SP, SP, X1  // Allocating frame:
3                          // SP -= length*8
4
5        MOV   X2, -1      // Use X2 as index
6        MOV   X3, SP     // Use X3 (current SP) as base
7        MOV   X4, -1      // Use X4 as the constant
8  In:   ADD   X2, X2, 1   // index ++
9        CMP   X2, X0      // Compare index and length
10       B.HS  Out         // If index >= length,
11                         // jump to Out
12       LSL   X5, X2, 3   // X5 = X2 * 8 as offset
13       STUR  X4, [X3,X5] // Store -1
14       B     In
15 Out:  ADD   SP, SP, X9  // De-allocating frame:
16                         // SP += length*8
17       RET
```

2.6.2.4 Leaf and Non-Leaf Procedures

In the example above, both `proc()` and `fun()` are called **leaf procedures**, because they didn't call any other procedures. The name could be more intuitive if you think procedure callings as a tree structure, where a callee is the caller's child. If they did call other procedures, they're referred as **non-leaf procedures**, and this is where we need to be careful.

Suppose we have changed the C code to the following:

```c
void proc(long length) {
    long arr[length] = {-1};
    arr[0] = fun(2,3);
}

long fun(long a, long b) { return a + b; }

int main() {
    proc(20);
}
```

In this case, `proc()` becomes a non-leaf procedure. What's special about non-leaf procedures?

The following assembly code is our first try:

```asm
/* -- Wrong code! -- */
proc:    LSL   X9, X0, 3   // length * 8
         SUB   SP, SP, X1  // Allocating frame:
                           // SP -= length*8
         MOV   X2, -1      // Use X2 as index
         MOV   X3, SP      // Use X3 (current SP) as base
         MOV   X4, -1      // Use X4 as the constant
In:      ADD   X2, X2, 1   // index ++
         CMP   X2, X0      // Compare index and length
         B.HS  Out         // If index >= length,
                           // jump to Out
         LSL   X5, X2, 3   // X5 = X2 * 8 as offset
         STUR  X4, [X3,X5] // Store -1 to memory
         B     In
Out:     MOV   X0, 2       // Parameter 1 <- 2
         MOV   X1, 3       // Parameter 2 <- 3
         BL    fun         // Call fun(2,3)
         STUR  X0, [X3]    // arr[0] = fun(2,3)
                           // (address 0x00000ff8)
                           // return address for fun()
         ADD   SP, SP, X9  // De-allocating frame:
                           // SP += length*8
         RET

fun:     ADD   X0, X0, X1  // a + b
         RET

```

```
28 _start:  MOV   X0, 20      // Pass 20 as parameter
29          BL    proc        // Call proc(20)
30          MOV   X0, 0       // Random instruction
31                            // (address 0x00001000)
32                            // return address for proc()
```

This is a very simple modification where we just move the three lines from the main procedure (`MOV`,`MOV`,`BL`) to `proc` (line 12 to 14), and added a `STUR` instruction (line 15) to store the return value of `fun()` to `arr[0]`. This is very intuitive, because that's how we modified our C code. What could go wrong then? Let's start walking through the instructions from line 13, right before we branch into procedure `fun()`.

On line 23, we branched to `proc()` using `BL` instruction. Remember what `BL` did is to store the return address (*i.e.*, the address of its next instruction in memory) into register `X30`. Let's assume when we go into `proc()`, `X30` has a value of `0x00001000`.

When we were at line 14, because we need to branch to another procedure, `BL` will put its return address—the address of the instruction on line 15—into `X30`, assuming it's `0x00000ff8`. You can see at this point, `X30` is **overwritten**. Surely we can correctly return from `fun()`, because `X30` stores the return address of `fun()`. However, there'll be a problem when we return from `proc()`, because on line 17, the value inside `X30` is still `0x00000ff8`, which is the address of the instruction on line 15.

The lesson learned here is, the machine doesn't remember whatever value you had in a register. If you overwrite `X30`, you overwrite it, and executing `RET` doesn't bring the old value back.

This comes down to just one problem, which is to avoid `X30` to be overwritten. Since `X30` is automatically set by `BL` instruction, the only way is for us to store `X30` on stack before using `BL`, and load it back to `X30` before `RET`.

The following code is a better version:

```
1  proc:    LSL   X9, X0, 3   // frame_size = length * 8
2           ADD   X9, X9, 8   // frame_size += 8
3           SUB   SP, SP, X9  // Allocating frame:
4                             // SP -= frame_size
5
6           /* ------------------------ */
7           /* Store X30 (LR) on stack  */
8              SUB   X10, X9, 8
9              STUR  LR, [SP, X10]
10          /* ------------------------ */
11
12          MOV   X2, -1      // Use X2 as index
13          MOV   X3, SP      // Use X3 (current SP) as base
14          MOV   X4, -1      // Use X4 as the constant
15 In:      ADD   X2, X2, 1   // index ++
16          CMP   X2, X0      // Compare index and length
```

```
17          B.HS  Out         // If index >= length,
18                            // jump to Out
19          LSL   X5, X2, 3   // X5 = X2 * 8 as offset
20          STUR  X4, [X3,X5] // Store -1 to memory
21          B     In
22  Out:    MOV   X0, 2       // Parameter 1 <- 2
23          MOV   X1, 3       // Parameter 2 <- 3
24          BL    fun         // Call fun(2,3)
25          STUR  X0, [X3]    // arr[0] = fun(2,3)
26                            // (address 0x00000ff8)
27                            // return address for fun()
28          /* ------------------------------ */
29          /* Load X30 (LR) back from stack  */
30             LDUR  LR, [SP, X10]
31          /* ------------------------------ */
32          ADD   SP, SP, X9  // De-allocating frame:
33                            // SP += frame_size
34          RET
35
36  fun:    ADD   X0, X0, X1  // a + b
37          RET
38
39  _start: MOV   X0, 20      // Pass 20 as parameter
40          BL    proc        // Call proc(20)
41          MOV   X0, 0       // Random instruction
42                            // (address 0x00001000)
43                            // return address for proc()
```

Let's walk through the changes we made. First of all, notice on line 2 we added eight more bytes to the size of the stack, because we know we need to store `X30` (can also referred as `LR` as in the code) on stack as well.

Next, we subtract eight bytes from `X9` and send it to `X10`. We use `X10` as the offset and `SP` as the base address to store `X30`, whic is at the bottom of the stack frame.

As discussed before, line 22 will change `X30`, but now we don't need to worry about it because it's copied on stack already. After we return from `fun()`, we restore `X30` back from stack, and thus successfully return back to the main procedure. Figure 2.11 shows the stack change.

☢ Caution!

Recall that `LDUR` and `STUR` will load/store eight bytes **from** the address we specified **upwards**. For example, assume `X0` has an address of a. The instruction `STUR X1,[X0]` will store the value of `X1` — the eight bytes — from a to $a + 7$. This can lead to some mistakes when using stack frames. Assume we have the following code for a procedure:

```
1  proc2:  SUB   SP,  SP,  24
2          STUR  X30, [SP, 24]
3          ...
```

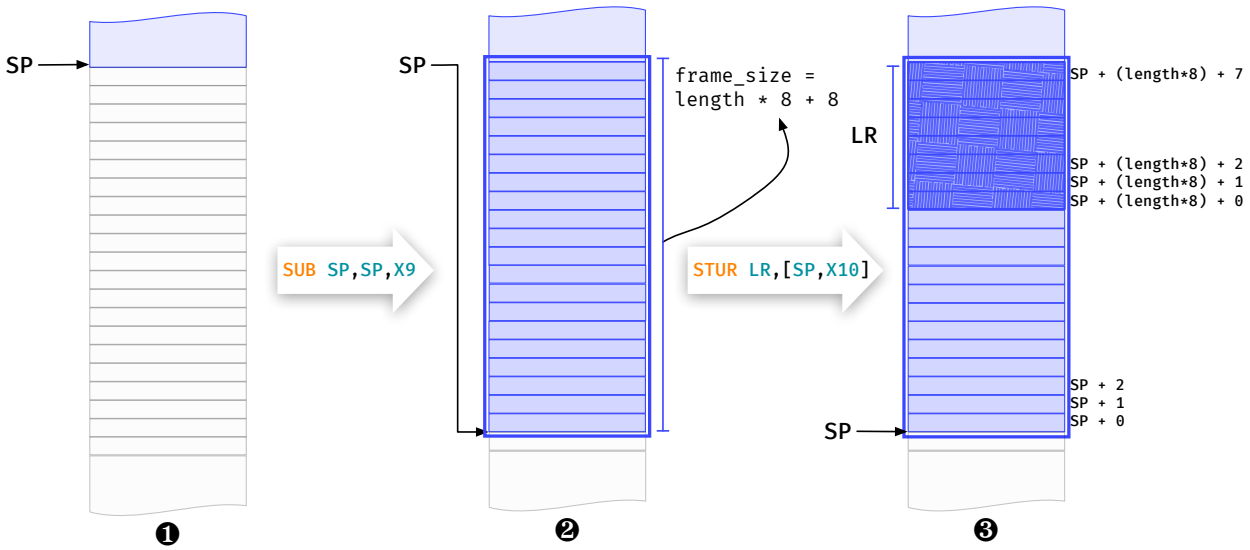


Figure 2.11: At point (1), SP was pointing to the stack top. We branched into proc, and calculated the size of the frame (line 1 and 2), and subtracted the size from SP, making it pointing to the top of the stack. The area between the old SP and the new SP is the procedure frame for proc. Using SP as base address, and X10 as offset, we used STUR to save X30 (LR) in the frame.

```
4         ...
5         LDUR  X30, [SP, 24]
6         ADD   SP,  SP, 24
7         RET
```

where we allocate a 24-byte frame, and want to save X30 to the bottom of the frame. This is wrong because STUR will store the eight bytes of X30 **from** SP+24 to SP+31, which is already out of the scope of the frame (the address of the frame runs from SP to SP+23). Then STUR will overwrite eight bytes outside the frame. The assembler will certainly not give you a warning, but you'll probably lost some important data. Same for LDUR. The correct way is to minus eight bytes from the size as the base address instead: `STUR X30, [SP, 16]` and `LDUR X30, [SP, 16]`. Figure 2.12 shows this idea.

Of course, if you're storing a word (4 bytes) to the bottom, then you'll need to subtract 4 bytes from the size, and use that number as the offset. No need to remember a formula; just make sure you calculated the addresses correctly.

2.6.2.5 Resolving Register Usage Conflicts

It is very common to have multiple nested procedure calls and even to call existing libraries. Because assembly is so simple it doesn't do any memory management or computing resource management for us, we'd have to be careful ourselves.

One very important problem is register conflicts. We can call as many procedures as we like, but there's only one set of registers. When the caller

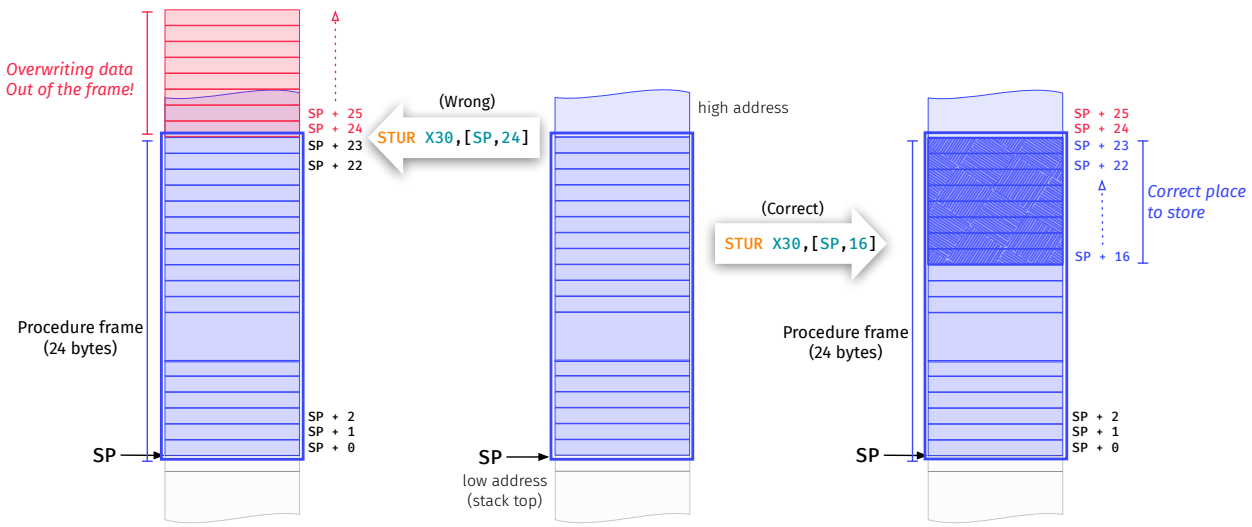


Figure 2.12: If the frame size is x, to store a double word to the frame bottom, we need to have an offset of $x - 8$, because `STUR` and `LDUR` start from low address and store/load to high address. Without subtracting eight bytes (left), we overwrite data that's out of the frame. On the right, the offset is 8 bytes smaller than the frame size, so we successfully put a double word (`X30`) to the bottom of the frame.

and callee wanted to use the same registers, we have to resolve the conflicts. From previous sections, we see that `X30` is a great example, where both procedures need it for return address. We resolved it by storing it on stack, and loading it back. This is the strategy we're going to use very often.

To avoid conflicts, both the caller and the callee are responsible for saving registers.

- **Callee:** the callee needs to store **callee-saved registers** and `SP` on stack, including `X19` to `X30`. Because the caller expect these registers are unchanged after procedure call, the callee needs to restore (meaning, `LDUR` back from memory to registers) these values before return;
- **Caller:** the caller needs to save **caller-saved registers** on stack, because there's no guarantee that the callee will not change the values of these registers. All the registers that are not callee-saved registers are caller-saved registers.

The following table shows us the work both the caller and the callee need to do around the point of procedure call and return.

	Caller	Callee
Before branching	Save `X0`...`X18`	× × ×
After entering callee	× × ×	Save `X19`...`X29`, `X30`
Before return	× × ×	Restore `X19`...`X29`, `X30`
After returning to the caller	Restore `X0`...`X18`	× × ×

Of course we don't have to save all the 19 caller-saved registers every time we branch; we just need to save those that we want to use later. If the data we want to use later can fit into registers `X19`...`X27`, then that's better, because it's not the caller's job to save them now.

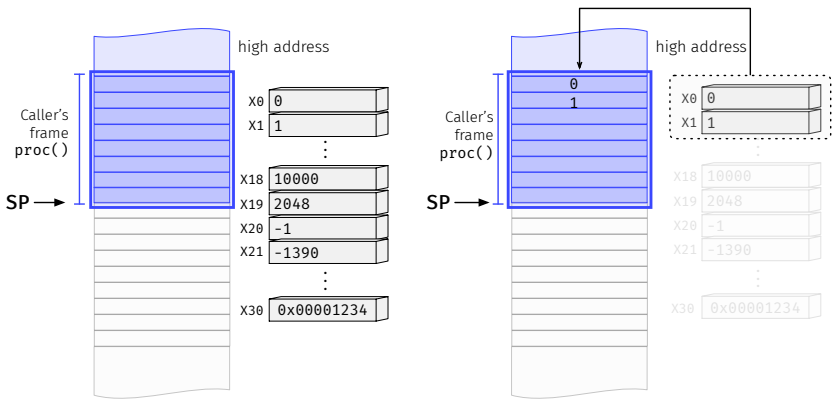


Figure 2.13: Because there's no guarantee that `X0`...`X7` will not be changed by the callee, before we branch to the callee, the caller needs to save **caller-saved** registers on stack. There's no need to save `X19`...`X29`, because we are certain that they wouldn't be changed by the callee. In other words, the callee will save and restore these registers.

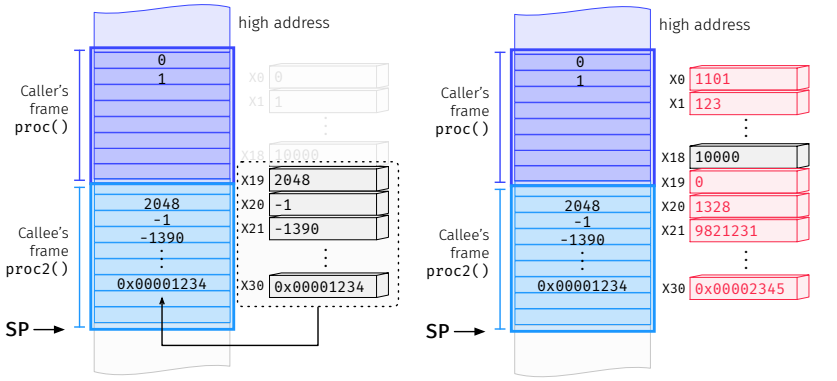


Figure 2.14: In the callee, to make sure the caller can still have the correct data in `X19`...`X29` and `X30` after return from the callee, we need to save them in the frame of the callee (left). Then during execution of the callee, it's totally ok to use all the registers without concern. On the right, all the changed registers are in red.

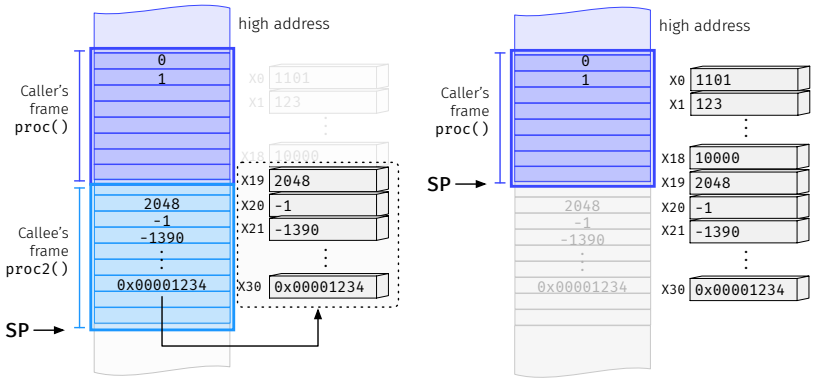


Figure 2.15: The callee is responsible for restoring the data in callee-saved registers, which creates an illusion to the caller that the data in `X19`...`X30` have never been changed. However, the callee is not responsible for all other registers. Therefore, if the caller wish to use the same data in `X0`...`X18` as before, it needs to save them before branching, and restore them after returning from the callee.

Let's look at the following example with illustrations. We assume procedure `proc()` calls `proc2()`. `proc()` has data stored in all registers `X0`, `X1` and `X19`...`X30`, and it expects that it can still use those data in these registers after calling `proc2()`. On the other hand, `proc2()` also needs to use those registers while executing. To resolve this conflict, before branch to `proc2()`, `proc()` will store `X0` and `X1` in its frame. It doesn't need to worry about other registers, because `X2`...`X18` are not relevant (assume `proc()` doesn't use them), and based on the standards `X19`...`X30` are callee's responsibility now. See Figure 2.13.

When inside the callee `proc2()`, the first thing it needs to do is to save those callee-saved registers (Figure 2.14). During execution of `proc2()`, it's safe to change those values.

Before return to the caller `proc()`, `proc2()` needs to restore `X19`...`X30` back to the registers (as promised!). Then to `proc()`, it looks like registers `X19`...`X30` have never been changed. Notice `X0` and `X1` are changed as well, but it's ok, because the caller `proc()` has saved them in Figure 2.13. Now it's the caller's job to restore these values if it wishes to use them again. Figure 2.15 illustrates this idea.

So why does this matter? Why do we create a convention and assing some registers as callee-saved registers? Currently, we are writing all our code in assembly, so with careful planning, register conflicts can be avoided to some extent.

However, imagine this: we need to link a procedure from a library. Do you know what registers that procedure will use? Apparently not. So how do you trust some registers after using a procedure that you don't know how it's implemented? ARMv8 created this convention about callee-saved registers like a contract, or a trust between us and other developers. Everyone developing in ARM, including you and me, needs to follow this convention or standard. It's like a promise from the developers: "*If you store your data in* `X19`...`X30` *before using my procedure, I can promise you that these data will not be lost. Other registers? Not my business.*"

Quick Check 2.6

Briefly answer the following questions on calling conventions:

1. How do we pass arguments into functions?
2. How are values returned by functions?
3. What is `SP` and how should it be used in the context of procedures in ARM assembly?
4. Which values need to saved by the caller, before jumping to a function using `BL`?
5. Which values need to be restored by the callee, before returning from a function?
6. In a bug-free program, which registers are guaranteed to be the same after a function call? Which registers aren't guaranteed to be the same?

2.6.3 Summary

Writing procedures needs careful planning, so we'll make a summary here to show general steps we can take when writing a procedure.

1. Calculate the procedure frame size (`frame_size`, must be multiples of 16), and subtract it from `SP`
2. Store registers `X19`...`X29` and `X30` on stack;
3. Do the procedure thing;
4. Put return value back to `X0`...`X7`;
5. Restore registers `X19`...`X29` and `X30` on stack;
6. Add `frame_size` back to `SP`;
7. `RET` back to the caller.

2.6.4 Reference

For detailed procedure call standards for ARMv8 architectures, there's no reference better than the official documentation released by ARM. Please go to Canvas and download the file *Procedure Call Standard for the ARM® 64-bit Architecture (AArch64)*.

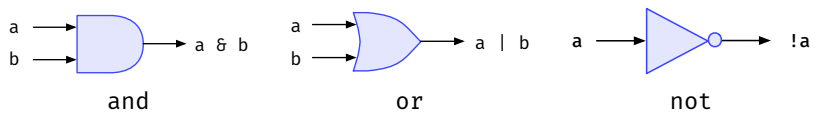
Microprocessor Design 3

The two most essential components of a microprocessor are registers and arithmetic logic unit (ALU), corresponding to two distinct functionalities—storing temporary data, and perform arithmetic calculations. In this chapter, we will start with logic fundamentals to look at small electronic devices used for building up a microprocessor, and then design a simple microprocessor that can executes a sequence of assembly instructions. After discussing the flaws of the design, we improve our model in terms of efficiency as well as preventing errors.

3.1 Fundamental of Logics

3.1.1 Logic Gates

The three most fundamental gates are `and`, `or`, and `not`. Each of them takes one input or two, and produces one output.We can build much more complicated logic using only these three gates.



In the figure above, each wire is either zero or one. In other words, each wire transfers one bit.

One thing we need to remember is, those gates instantly respond to signal changes. Assume at one moment, we have `1` (high voltage) and `0` (low voltage) passing through the `and` gate, and the output is `0`. If at some point we change the low voltage input to high voltage, the output will instantly change to `1`. [1]

In Figure 3.1, we show a time series where `a&b` changes as `a` and `b` change their values as the inputs of the `and` gate. At timestamp ①, inputs `a` has high voltage (consider it as signal `1`) while `b` has low voltage (as `0`), and therefore `a&b` is `0`. At the end of ②, input `b` raised its voltage to `1`, so `a&b` started reflecting this change to `1`. The small amount of delay between ② and ③ is called **rising delay**. Similarly, the delay between ④ and ⑤ is called **falling delay**. This amount of delay is almost neglectable, so we roughly consider the change of the output of these gates is almost instant.

1: Strictly speaking, there's a tiny teeny amount of delay, but that amount of time can be neglected without any problem.

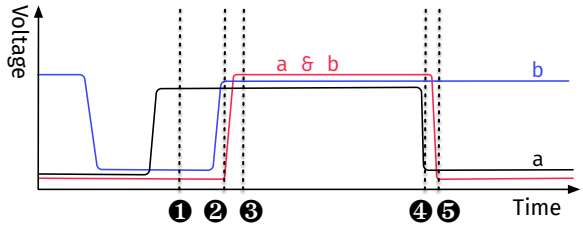


Figure 3.1: a and b are the two inputs of the and gate, and a&b is the output. The and gate constantly and almost immediately reflects the change of the input, with small amount of delay which is neglectable.

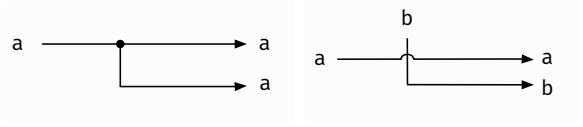


Figure 3.2: On the left, signal a has been branched into two; on the right, a and b are separate signals without relations, indicated by a line hop.

Line Notations

You probably already know this but just a small refresher — as in Figure 3.2 left, the small dot means the two wires come from the same wire, so they have the same signals. On the right, the two wires have no relations as there's "line hop".

3.1.2 Combinational Logic

Now that we have the three fundamental gates, we can use them to build more complicated logics. When we combine these gates and the signals are in one direction without loop, it's called a **combinational logic**. Because there's no loops in the combinational logics, the signal changes are almost immediate, and will respond to input changes constantly.

3.1.2.1 Comparison

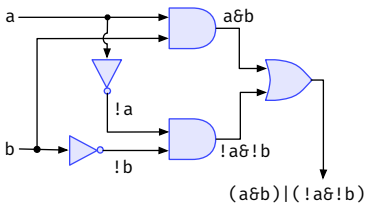


Figure 3.3: The combinational logic for comparing two bits. When the two bits are equal, the output is 1; otherwise it's 0.

Let's create a very simple combination logic, where we compare if two bits are the same. If they are the same, we output a signal of 1; otherwise a 0. The combinational logic for bit equality is shown in Figure 3.3. Both a and b are input, and using a truth table we can easily verify that when a==b, output is 1; otherwise it's 0.

Now imagine we want to compare if two double words (64 bits) are equal. One way to do this is that for each second, we pass one bit of each number to the bit equality logic as inputs, and take notes on the output. After one minute and four seconds of waiting, we got 64 bits of outputs. If all of them are 1s, we know these two numbers are the same; otherwise it's different. A major flaw of this method is we have to wait for 64 seconds! So why don't we compare all 64 bits at a time? Figure 3.4 shows this idea.

In Figure 3.4, we use a[i] to denote the i-th bit of the double word a. On the left, the two sets of parallel wires transfer a and b, one bit each wire. These two are the **buses**. Moving towards right, we see each bit from each number is directed to a corresponding bit-equality logic, and produces the output eq[i]. Then all the outputs from 64 bit-equality logics are pushed

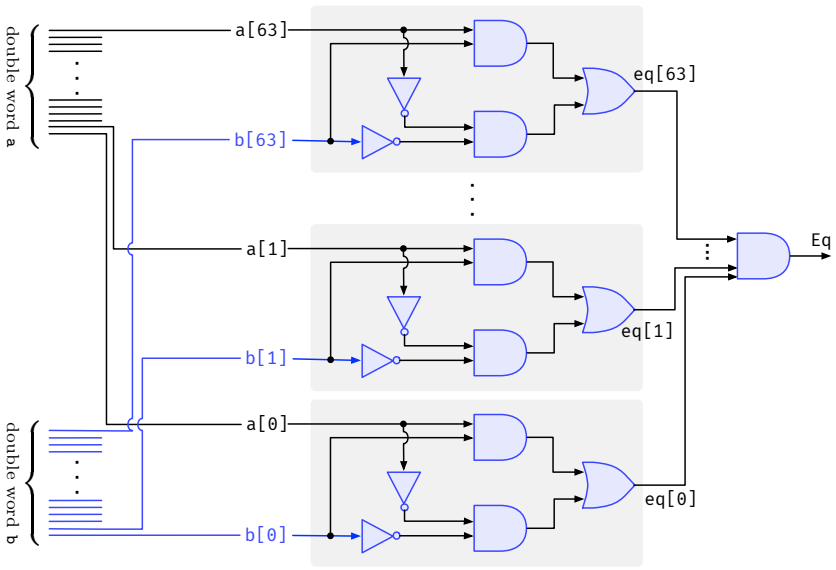


Figure 3.4: The parallel sets of wires on the left are called buses, where each wire transfers one bit of data, and all the wires transfer data at the same time. To make the graph clearer, we made lines from input `b` blue. Each pair of input bits uses the bit equality logic in Figure 3.3 to compare.

into a final `and` gate where a single bit `Eq` (either `1` or `0`) is generated.

3.1.2.2 Selection

Another simple but important combinational logic is to select one of two (or multiple) inputs as an output. Consider it like as faucet where we can get hot and cold water. The "inputs" are hot and cold water, but we can only have one output, and it's either one of them. The logic that chooses one of the inputs as the output is called **multiplexor** (usually denoted as `MUX`). Let's start with the bit multiplexor.

In Figure 3.5, we show the logic of a bit-multiplexor. In addition to input signals `a` and `b`, we also need to introduce a **control signal** `s`. This control signal is like a switch that controls which input can pass through the multiplexor and become the output. Usually, when a control signal is `1`, we call the signal is **asserted** or **set**; otherwise **deasserted** or **clear**. By using a truth table, it's not difficult to notice when `s == 1`, the output is `b`; otherwise it's `a`.

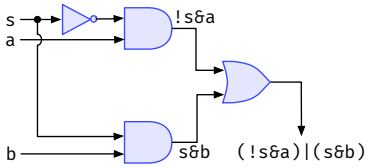


Figure 3.5: The combinational logic for selecting one of the inputs as the output. Input `s` acts as a "switch", or "control". When `s == 1` (asserted), input `b` passes through the multiplexor; when `s == 0` (deasserted), input `a` passes through.

Notice in this multiplexor, technically we have three inputs: `a`, `b`, and `s`, but we call `a` and `b` the input in the sense of "data input", while `s` "control signal input". In the future, we will usually use "input" to refer to data input, but its actual meaning should be clear based on context.

We show the logic for selecting one of the two double words in Figure 3.6. Similar to the equality logic, here for each bit of the input, we use a multiplexor to select. One thing we need to notice is in this example, the control signal has the same value for all the bits of each double word, because we want to select all bits of a double word, so they should have the same control signal. The output, instead of being just one bit, contains 64 bits, and they are equal to either `a` or `b`, depending on the control signal `s`.

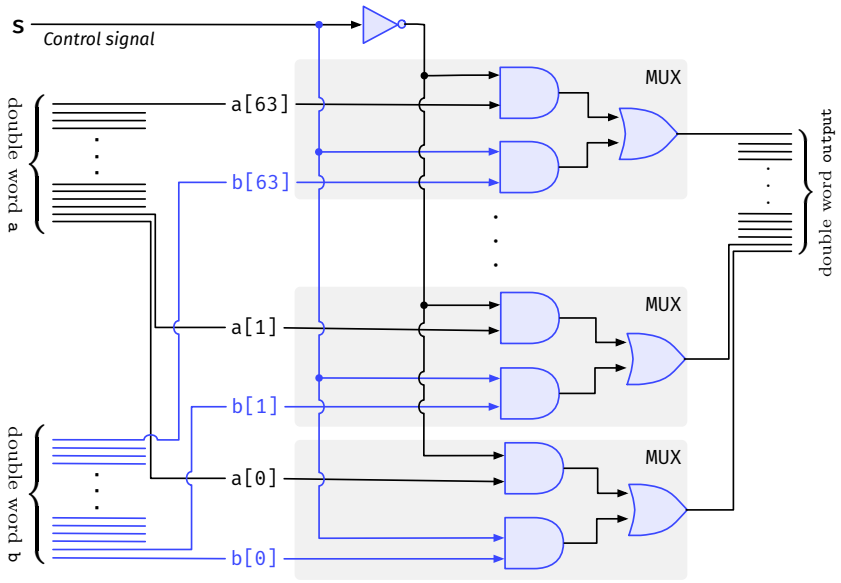


Figure 3.6: The inputs `a` and `b`, as well as the output, are all 64-bit double words. The same control signal controls all the bits of an input, which makes the logic choose one of the inputs for every bit, and thus choose one double word to pass through. We highlighted the wires for input `b` and its corresponding control signal wires.

In Figure 3.6, we only need one bit of control signal. In some cases we need more than one bit, and thus all the control signals together form a **control bus**.

N-Way Multiplexor

If we need to select more than two inputs, we'd need to create a N-way multiplexor. Notice that when there are N inputs, one bit of control signal is not enough apparently. For example, if $N = 4$, we'd need two-bit control signals, so that `00`, `01`, `10`, and `11` will choose one of the inputs as the output. In general, the number of control signals can be calculated as

$$S = \lceil \log_2 N \rceil \tag{3.1}$$

where $\lceil x \rceil$ is the ceiling function that takes the nearest integer above x.

3.1.2.3 Arithmetics

Recall back in Section 1.2.1.2 we introduced Arithmetic Logic Unit (ALU), the core part of the processor, as it does all the calculations. One of the functions an ALU can operate is addition, *i.e.*, adding two inputs together. Since ALU itself is also combinational, in this section we'll see how we can use those simple logic gates to build a logic that can perform addition operations.

The logic for adder is also built on logic operations. Without thinking about how the gates are orgniazed, let's start with a simple truth table first. Assume the two inputs are `x` and `y`. When doing addition, we need to add a carry-in flag, called `cin`. Since both `x` and `y` are one bit data, if the addition result takes two bits, say `z[1]z[0]`, the leading bit `z[1]` will

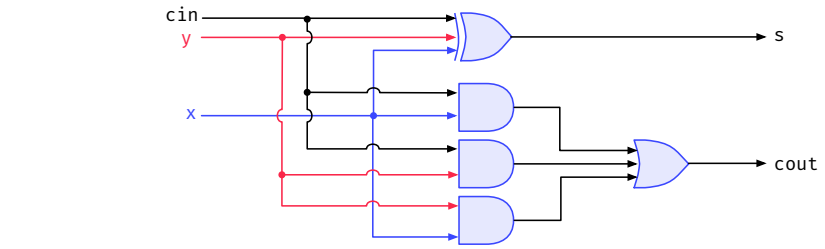


Figure 3.7: Bit adder, where input `y` is marked as red, `x` as blue, the carry-in signal `cin` as black.

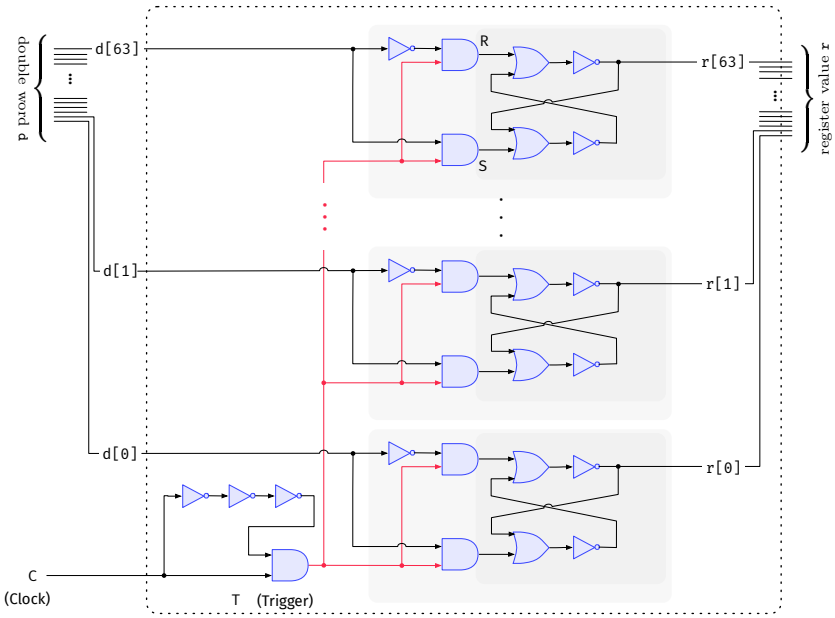


Figure 3.8: Full 64-bit adder using the bit adder design from Figure 3.7. From bit 0 to bit 62, each adder's carry-out flag `cout[i]` will be used as carry-in flag for bit `i+1`'s adder.

be the carry-out flag `cout`, while the last bit `z[0]` is the addition result, denoted as `s`.

Once the input and output are determined using the truth table, the rest of the work is simply to design a combination using all the possible gates, to make sure it can produce the correct output given each input. As long as it can produce correct values, any combination is valid, though we surely favor simpler designs. Figure 3.7 is one of the possible designs, where we use an `xor` gate (the `or` gate with a curve).

Similar to how we built up a 64-bit multiplexor, when there are two double words `a` and `b`, we align the bits for the two numbers, and send a pair of bits into a bit adder. See Figure 3.8. What's different here is, from LSB to MSB, the carry flag from bit `i` will be also used as the carry flag for bit `i+1`, just like calculating by hand. The first carry flag can simply just be zero, and the carry flag produced by MSB, *i.e.*, `cout[63]`, will be sent to CSPR's carry flag. (Still remember condition codes?)

x	y	cin	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

3.1.3 Sequential Logic

When introducing combinational logic, we are clear that it doesn't contain any cycles: the output of a gate will not become its input at some point. We

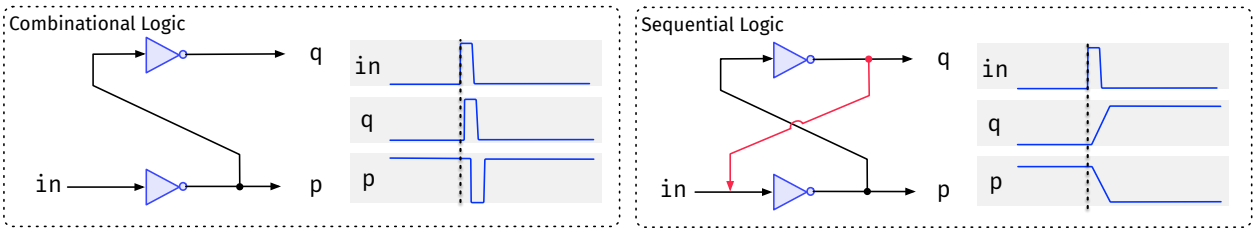


Figure 3.9: The top shows a combinational logic, whereas the bottom shows a sequential logic. In the combinational logic, both outputs `p` and `q` respond to the change of input `in` almost instantly, and thus we are not able to "store" the output. In the sequential logic, however, one temporary change in the input `in` will trigger the permanent change in the outputs, making them stay, and thus to be "stored".

also emphasized that combinational logic will constantly change according to its input signals. The question is, how can we *store* data, instead of simply calculating data? In this section we introduce **sequential logic**, which exactly serves this purpose.

Figure 3.9 shows us a very simple sequential logic, called **bistable element**, where we compare it with a similar combinational logic. On the top, the logic is acyclic with two outputs, `p` and `q`, and an input signal `in`. As shown in the time series on the right, when we give `in` a high voltage pulse, both `p` and `q` respond to the input with small amount of delay, but soon change back. Therefore, the input `in` is temporary, and so are the outputs. What if we want the outputs stay where they are? This means we want to *store* the outputs.

At the bottom of Figure 3.9, we added a branch from `q` back to the input of `p`, making a loop. This is not combinational anymore; it's sequential instead. On the right, we see even if we give a small and temporary high voltage to `in`, both `q` and `p` stay where they are and keep the change after the signal from `in` disappears. Apparently, this is because of the loop, where the output of `q` serves as the input of `p` so it doesn't rely on signal `in` anymore.

For a real life example, think about a faucet again. The combinational logic is like the automatic faucet that can senses your hands. If your hands are close, there's water; if you take your hands away, it stopped. The faucet constantly respond to our "input" — hands. What if we want the water keep flowing? We just change to a regular faucet, where we only need to turn on the water manually once, and it'll keep flowing until we manually turn it off. In this scenario, the regular faucet only responds to our "input" once, and will keep the change.

This bistable element is very simple but shows an important idea of sequential logic. One flaw of the one in Figure 3.9 is, however, how can we change the output `p` and `q` back?

3.1.3.1 SR Latch

SR latch, or Set-Reset latch, allows us to change the output anytime, as shown in Figure 3.10. Here we have two inputs `S` and `R`, representing "set"

and "reset" respectively. The outputs are denoted by `Q+` and `Q-`. It's not difficult to notice that `Q+` and `Q-` always have the opposite values.

In the time series on the right, we changed the output `Q+` to 1 by giving `S` a temporary high voltage at timestamp (1). At timestamp (2), if we want to change `Q+` back to 0, we can give `R` a high voltage (*i.e.*, "reset" the value of `Q+`). All the changes in the outputs are permanent, unless we trigger either `S` or `R`.

SR latches have four states, three of which are shown as the three timestamps in Figure 3.10: (1) setting, (2) resetting, and (3) latched or stored. The state where we give high voltage to both `S` and `R` is called *metastable*, which usually causes error.

We can write a truth table for SR latch as well, but it's a little bit different where we let `q` denote the actual value (either 1 or 0) of the output of `Q+`. See the table below:

S	R	Q+	Q-	State
0	0	q	!q	Latched (Stored)
0	1	1	0	Resetting
1	0	0	1	Setting
1	1	–	–	Metastable / Error

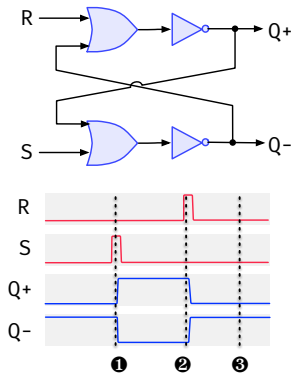


Figure 3.10: A simple SR latch and an example of time series. Time (1) is the state of setting, (2) for resetting, and (3) for latched. The temporary change in either `R` or `S` will make the change in the output stay, and thus to be stored.

As we see, unlike the truth table we are familiar with where all the outputs have determined values of either 0 or 1, the state of latched has `q` and `q!` in it. This is because the output depends on its actual value before switching to this state, instead of the inputs `S` and `R`.

3.1.3.2 D Latch

We're getting closer and closer to the actual implementation of storage devices in microprocessors! SR latch is great because it allows us to change the output arbitrarily and make the change stay. Two flaws of SR latch, however:

1. `S` and `R` are like switches; where do we send and store actual data?
2. `S` and `R` can be used at any time, but when there are many SR latches, how can we make sure they are synchronized, or on the same page?

These questions bring us to a better design, called **D latch**, where D stands for data. D latches explicitly addressed the two flaws mentioned above: a new input is used for data, denoted as `D`, while another input is used for a clock, denoted as `C`. See Figure 3.11.

The clock `C` controls when the input data `D` can pass through and cause changes in the output `Q+`.[2] At timestamp (1), `C` gives a high voltage, which brings the D latch to a state called **latching**. At this moment, `Q+` will change its value according to the input data `D`. As long as `C` stays at high voltage, `Q+` will always respond to the change of `D`.

2: Note `Q-` always outputs the reverse of `Q+` so we'll simply ignore it.

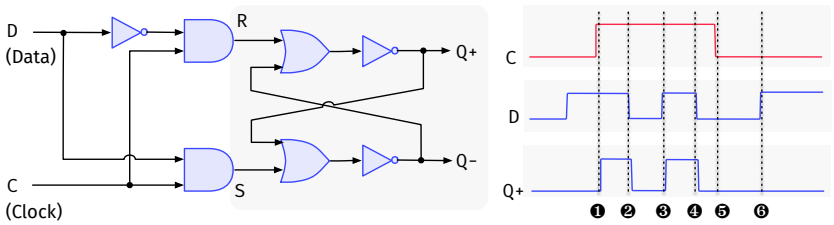


Figure 3.11: D latch has a clock `C` to control when the data `D` is allowed to pass through and to cause change in the output `Q+`. When `C` is 1, the status is called "latching", where output `Q+` responds to the change of the input data `D`. When `C` is 0, the status is "storing", and `Q+` stays/-stores the value regardless of changes of input `D`.

At timestamp ⑤, clock `C` falls back, bringing the D latch to a state of **storing**. This means that we have stored the input `D` in `Q+`. After this, as long as `C` stays at 0, no matter how we change `D`, `Q+` will not change (*e.g.*, see timestamp ⑥); it's storing the value of input data `D` at the moment that the clock falls.

We can similarly write a truth table as the one on the side where `d` can be either 0 or 1, and the value of `q` depends on the value of `d` at the moment when clock `C` drops.

C	D	Q+	Q-	State
0	d	q	!q	Storing
1	d	d	!d	Latching

3.1.3.3 Flip-Flops

Last stop before we get to the real thing in microprocessors! D latches allow us to use a clock to control the output with a data input, which is good, but notice that as long as `C` stays at high voltage, `Q+` will change based on `D`. This is not expected, because any interference of the input signal will affect the output, making it unstable to store. Imagine you send an input data of 1, and during latching mode, the signal of input becomes unstable and so does the output `Q+`. If we cannot change the frequency of the clock, how can we make sure during latching state the output `Q+` stays stable?

The solution is **edge-triggered latches**, or **flip-flops**, where the output will change *only* when the clock is on the rise edge. The output will not change even when the clock is 1.

We use Figure 3.12 to show the logic of a flip-flop and an example of time series. The major change is we added a trigger `T`, which is essentially just an `and` gate. Both two inputs of the trigger come from clock `C`. If you look closely, after passing through three `not` gates, one input is `!C`, while the other is `C`, so the trigger will be `C & !C = 0` eventually. Then why do we want to create a trigger this way?

Remember we said we only want the output changes when `C` is on the rising edge. In this case, trigger `T` controls if `Q+` will change with `D`, so that means we only want `T` to be at high voltage at a very short amount of time, *i.e.*, when the clock rises. So how can we create such a small pulse to allow `Q+` stores `D` and stays stable afterwards? We just need to make sure `T` can be 1 when clock rises, and quickly swtich back to 0 eventually.

From the beginning, we mentioned that logic gates will constantly respond to input changes, but *with small amount of delay*, meaning the more gates a signal needs to pass, the longer delay it'll cause. There you go! See when `C` rises, one of its two branches will quickly reach to the trigger, at which

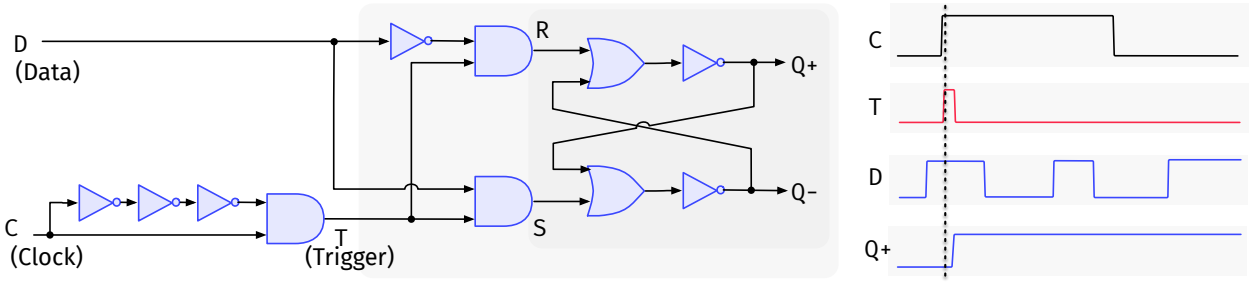


Figure 3.12: An edge-triggered latch, or a flip-flop, where when `C` rises, the trigger `T` will temporarily rise to high voltage, allowing `Q+` store the value of input data `D` at that moment. Afterwards, `T` drops back down, and no matter how input `D` changes `Q+` stays stable.

moment the trigger will rise to high voltage. The other branch of `C` will arrive shortly but with a delay, because it has so many gates to pass through. It will, however, eventually arrive, and thus brings the trigger back to low voltage. This gap of arrival of the two inputs gives us the chance to change `Q+` and store its value, and make it stay stable afterwards.

3.1.4 Registers

Finally! It's time to talk about the real thing! In microprocessors, the hardware we use to store data is registers, our old friend from assembly. Now that we know one flip-flop can store one bit of data, storing a double word is just too straightforward. In fact, as shown in Figure 3.13, a register is just a group of flip-flops where each of them stores one bit. Notice that all the flip-flops in the register are controled by the same clock signal. After all, you don't want to store several bits first and wait until next rising edge of the clock to store other bits.

In ARMv8 architecture, we have 32 general purpose registers, all controlled by one clock signal. All registers are encapsulated in a group called **register file**, and in our architecture one microprocessor has one register file. Figure 3.14 shows an illustration of a register file.

Each register file has two **read ports** and one **write port**. Here based on Figure 3.14, we declare the signals with "variable-style" names, so we can refer back to them later.

On the write port, we have three input signals: `RegDataW` for the actual data we want to store into the register; `WriteReg` for the destination register number, *i.e.*, where do we want to store `RegDataW`; and `RegWrite`, used for indicating if we want to write to a register or not.

Input `WriteReg`, a 5-bit data, is sent to a decoder, where the output is ad-hoc. For example, if `WriteReg = 00111b`, the output of the decoder will be all zero except the wire transferred to register `X7`, because `00111b` is `7` in decimal. Because of the `and` gates, only `X7` will be written once `RegWrite` changes to `1`, meaning we do want to write to the register.

Each register has two inputs, `C` and `D`, corresponding to the inputs in Figure 3.13. Notice that the `RegWrite` is connected to the input `C` of the registers.

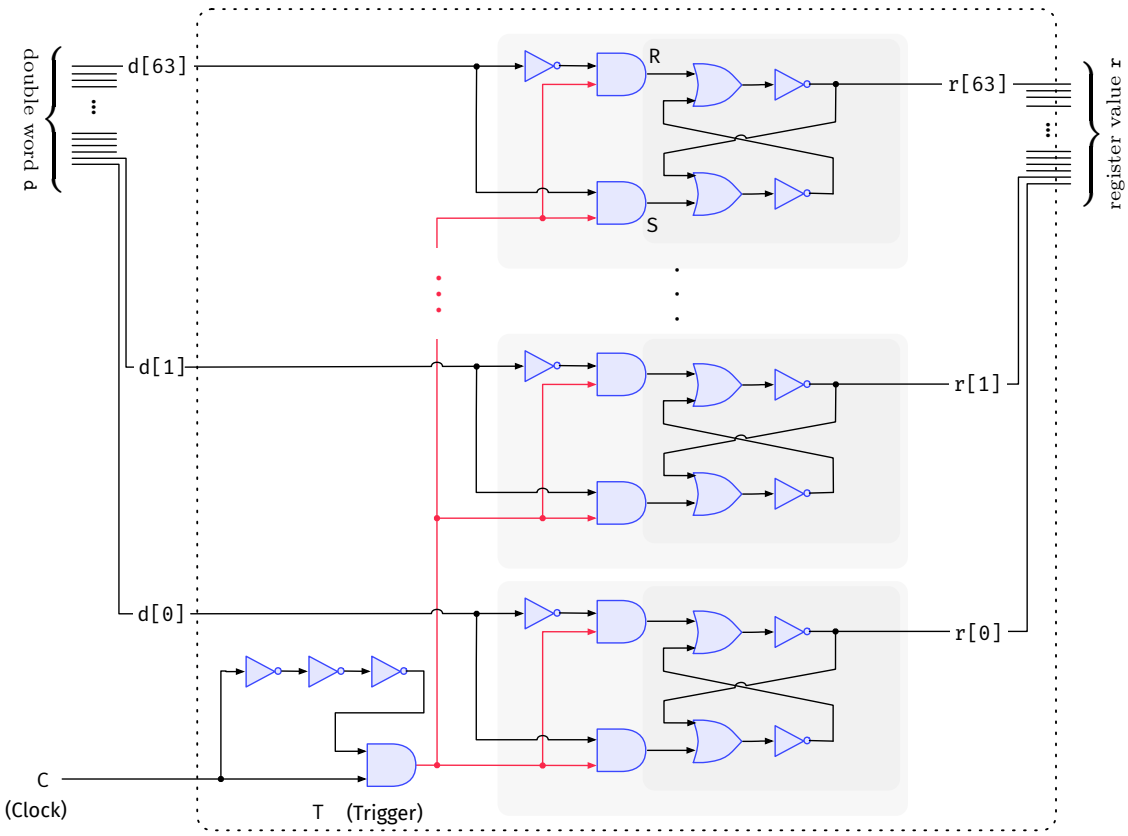


Figure 3.13: A register implemented using edge-triggered latches. One latch can store one bit of data, and all the 64 bits will be updated all together when the clock rises.

In fact, writing to a register doesn't happen at any time — *it only happens when the clock rises*, because it's a sequential logic.

On the right side of Figure 3.14, we have two read ports. The two register numbers, `ReadReg1` and `ReadReg2`, are used as "selectors" to select data from corresponding registers. The data are denoted as `RegData1` and `RegData2`, respectively. *Reading registers is more like a combinational logic*, meaning as soon as `ReadReg1` and `ReadReg2` receives changes, the corresponding data will be read almost instantly. This is different than writing to a register, since the read port has no control wire that connects to the clock.

3.1.5 Memory

Random Access Memory (RAM), or simply memory, also uses sequential logic to store data, even if it's not inside the CPU. Figure 3.15 shows an illustration for memory.

In practice, memory has three buses—data, address, and control buses. The data bus is bidirectional: it can send data to memory, or read data from memory. This means, during access to the memory each time, we

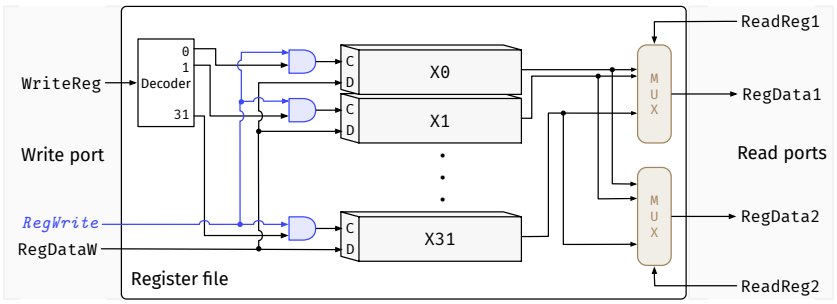


Figure 3.14: A little more detailed register file.

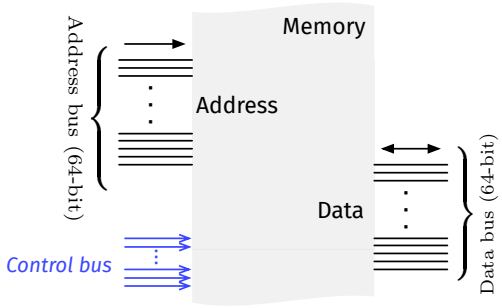


Figure 3.15: Control bus and address bus are unidirectional, while data bus is bidirectional.

can only either read data or write data—**we can't do both at the same time!**

The address bus is used for a memory location. During reading, the data at the address on address bus will be put on data bus and sent out. Similarly, during writing, the data on the data bus will be sent into memory location specified by the address bus. Either reading or writing is controlled by the control bus. The width of the control bus — the number of bits — depends on specific architecture. In our design, we only need two bits.

3.1.6 Summary

The most essential components of a microprocessor have been introduced in this section. In sum, each microprocessor has two basic functionalities—computing and temporary storage. The component used for computing is arithmetic logic unit (ALU), implemented using combinational circuits. ALU can perform basic arithmetics, and we showed the operation of addition in Figure 3.8. Registers are used for temporary data storage, implemented by sequential circuits (Figure 3.13). Writing data to a register is controlled by a clock, and only happens when the clock signal has a rising edge. Reading data from registers, however, is similar to combinational logic, and can happen any time.

3.2 From Assembly to Machine Code

Before we dive into the real construction of our CPU, we need to consider how we can first let it recognize our program, or instructions. Surely *we* understand `ADD X0,X0,X1` is to add `X1` to `X0`, but those digital circuits

only recognize high and low voltage, or at least only 0s and 1s. The first step for us, therefore, is to translate our text code (assembly) into digital signals (binary 0s and 1s) so that they can be pushed into the circuits and perform operations. Once the circuits finished responding to the signals, we interpret the resulting 0s and 1s back to know what's going on.

Each instruction is unique: they have different operands and operators; but some instructions have similar attributes: they perform the similar operations. For example, both `ADD X0,X0,X1` and `ADDS X20,X21,0xff14` are calculating the sum of two numbers, and storing it to a register. When translating those instructions into binary, we need to make sure they are distinct enough so the machine can perform the job requested precisely, but also somehow similar to reduce the repeated work on the hardware design. This "translation" from text assembly instruction to binary machine code is called **encoding**.

Since we are using ARM assembly, we will look at the encodings designed by ARM, so no need to design our own. In the following, we will only study encodings of most frequently used instructions. They are sufficient to serve the purpose of the discussion of CPU design. We also removed and modified some of the fields in the encodings to make them easier and more straightforward. To see the complete sets of encodings, you can visit the ARM documentation. [3]

3: See https://developer.arm.com/documentation/ddi0602/2022-03/Base-Instructions.

Each ARM assembly instruction takes four bytes (32 bits), and can be roughly separated into two fields. The leading bits are called `opcode`; which is unique to different mnemonics. The rest of the bits are used for operands, such as encoding register read/write numbers, immediates, addresses, *etc.*

3.2.1 Arithmetic/Logic Instructions

Let's start with the ones we're most familiar with—arithmetic and logic instructions.

3.2.1.1 With Registers

The first case is all the sources are registers, including `ADD[S]`, `SUB[S]`, `AND[S]`, and `ORR`. Looking at these instructions, we notice they all take three register operands: destination `Rd`, source 1 `Rn`, and source 2 `Rm`. Thus, we show their encodings in Figure 3.16.

It is straightforward that all the register files take five bits, since we have 32 general purpose registers which can be encoded in five bits. Note that here `Rm` refers to the register *number* or *ID*, instead of the data inside of them. The bits `15:10` are all zeros for all instructions. [4]

4: In fact, bits `15:10` contain 3-bit of `option` and a 3-bit immediate. To keep things simple and focused, we don't really use or care about the fields, so we can simply let `option` in all such instructions be `111`, while immediate be `000`. As to why we use these two values, if you're interested, feel free to ask me, or visit ARM documentation.

It's not difficult to find some patterns in the `opcode`. For example, it is clear that bit 30 is to indicate which operation we want: `0` for addition, while `1` for subtraction. Also, bit 29 is `1` if we want to set condition codes, or `0` otherwise.

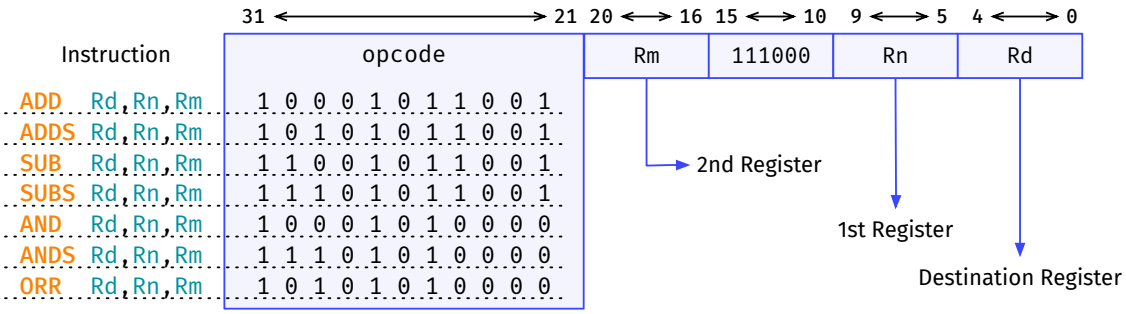


Figure 3.16: Encodings of arithmetic and logic instructions with register operands.

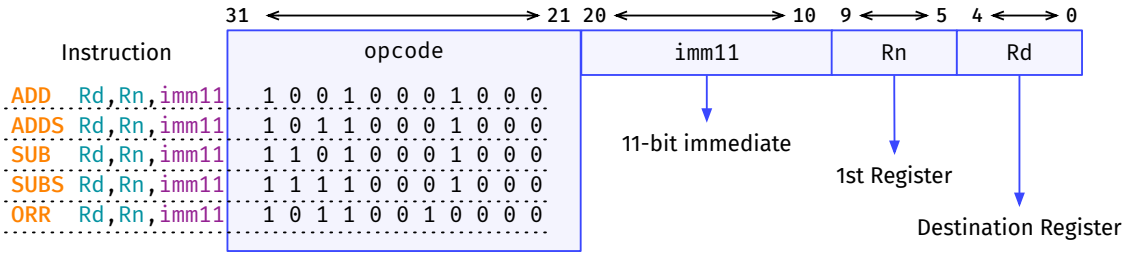


Figure 3.17: Encodings of arithmetic and logic instructions with register operands and immediates.

Quick Check 3.1

Remember `CMP Rn,Rm` is to compare two registers, and set condition codes. What it actually does, is to perform subtraction between `Rn` and `Rm`, and send the result to `XZR` (register `X31`). In other words, it's just an alias for instruction `SUBS XZR,Rn,Rm`. Please translate `CMP X1,X0` into machine code based on the discussion above.

3.2.1.2 With Immediates

The general structure of instructions with immediates are still the same—a few bits of `opcode` followed by operands, as in Figure 3.17. Notice `Rd` is the destination register, `Rn` the source register, and `imm11` is a 11-bit immediate number.

Quick Check 3.2

When looking at machine code, you found a binary sequence:

```
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0
                    0 1 0 1 0
```

What assembly instruction does it represent?

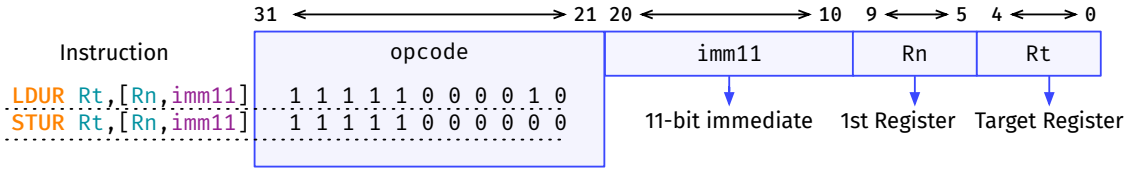


Figure 3.18: Encodings of memory accessing instructions.

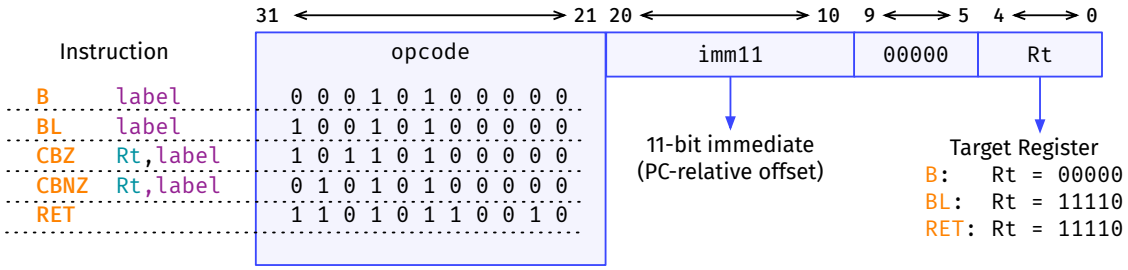


Figure 3.19: Encodings of branching instructions.

3.2.2 Memory Accessing Instructions

Here we mainly focus on `LDUR` and `STUR` with only 64-bit registers supported. In these two instructions, as shown in Figure 3.18, the 11-bit immediate number indicates offset, while `Rn` is the base register. Since `Rt` could be source or destination register, we call it "target register".

3.2.3 Branching Instructions

Branching instructions follow the same format. To keep our instruction set consistent with the datapath, we only consider the instructions shown in Figure 3.19.

Notice the immediate number encoded in bits `[20:10]` are PC-relative offset. For example, assume current instruction is `B L1` at address of `x`. The instruction labeled as `L1` is at address of `y`. Thus, the `imm11` field of `B L1` is `y-x` (and then zero extended to 11 bits). This is done automatically by the assembler.

For bits `[4:0]`, instruction `CBZ` does have `Rt` in it, so it'll just be the register number. For `B`, since no register is involved, we simply put all zeros in `Rt`.[5] Instructions `BL` and `RET` use `X30` by default, so `11110` is hardcoded into `Rt`.

5: Note that technically all zeros in `Rt` represent register `X0`, but this instruction doesn't even need to read a register, so we're fine.

3.3 A Sequential Datapath

3.3.1 Preliminaries

Sometimes a line of code speaks louder than a thousand words, so to better illustrate the idea of datapath, in the following we will use a "Python-like"

language. It has similarities as to syntax to Python, but doesn't necessarily follow correct syntax. Our focus is not writing a correct Python program, but to make the datapath clearer.

Notations and Description Language

We use `R[x]` to denote the 64-bit data inside register `x`, while `M[a]` the data at address of `a` inside memory. For a multibit data `D`, `D[a:b]` indicate bit `a` to `b` (inclusive). For example, `R[0][5:2]` is the 2nd to the 5th bits inside register `X0`, while `M[R[1][31:0]]` is the data at an address indicated by the lower word of data inside `X1`.

We often use a compact `if-else` statement, such as `data = x if b else y` where `data` is `x` if `b` is `True` or `1`; otherwise `data` is `y`. This can also be nested, such as `data = x if b else y if a else z` : if `b == 1`, `data = x`; else if `a == 1`, `data = y`; otherwise `data = z`.

Wirings

In the illustrations of datapath, we have two types of wirings. Blue lines are to transfer **control signals**, usually one bit of data, but could be more. Control signals are used to guide CPU to perform specific operations, and closely related to the mnemonics/operators of each instruction.

Black lines are used for **data signals**, where real data (such as operands) are transfered. Data wires are usually parallel lines (buses) where each line transfers one bit, and all bits are transferred all at once. For example, data buses and address buses can transfer 64 bits at a time. To simply the drawings, we use only thin lines to indicate such signals or buses, but it should be clear that they have wider bandwidth of data. This way, we focus on the *flow* of data in the system by removing details about actual width of those wires.

3.3.2 Stages of an Instruction

Figure 3.20 shows our first attempt at building the datapath, where we'll run one assembly instruction at a time. We call this model **sequential implementation**. Note that it has nothing to do with the sequential logic/-circuits we discussed in Section 3.1.3. This is absolutely over-simplified and inefficient, but it shows us the most important ideas of designing a processor.

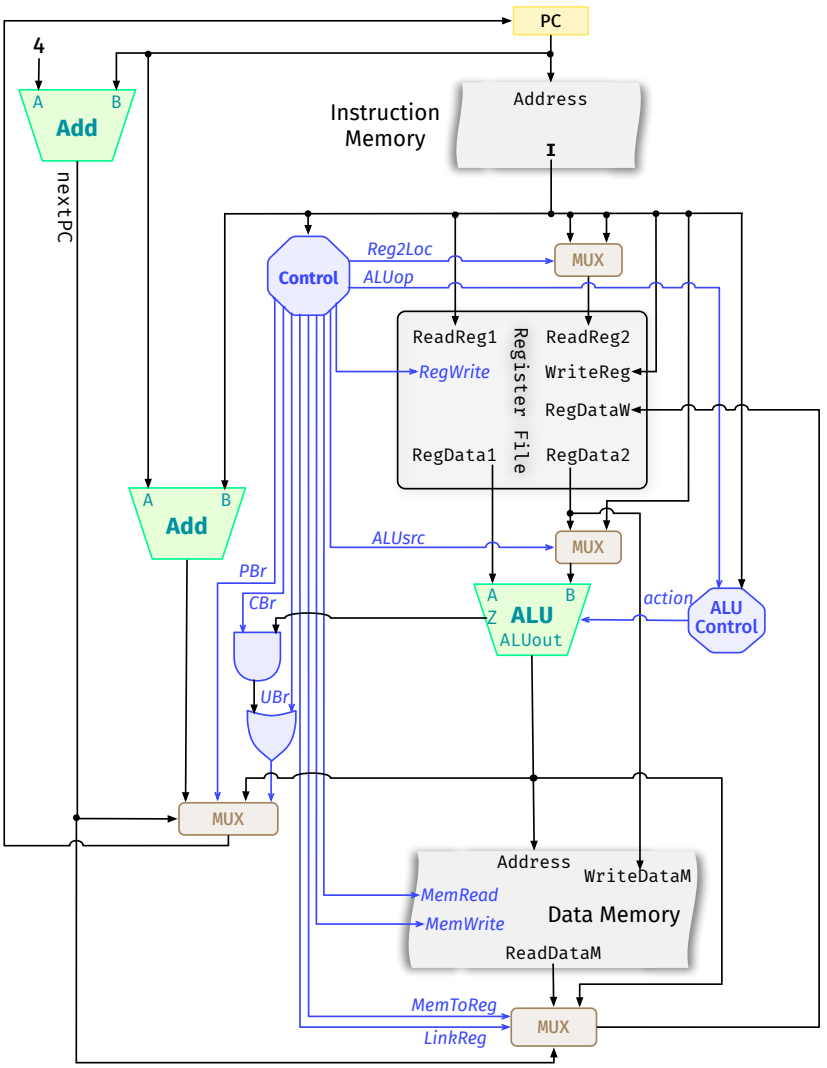


Figure 3.20: A sequential implementation of datapath. Black lines are data signals, while blue lines are control signals.

In Figure 3.20, we separate memory into instruction and data memory, only for illustration purposes. It should be clear that there's only one memory, where both instructions and data are stored together but in different regions. Instructions are in `.text` segments, and regular data are in `.data`, `.bss`, or stack area of the memory.

Each instruction, from being obtained from memory, to finished executing, will follow the five stages as it passes through the datapath:

1. **(IF) Instruction Fetching:** the instruction is obtained from memory;
2. **(ID) Instruction Decoding:** the fetched instruction will pass different fields to different data signals, and the `opcode` will be used for converting into control signals. Register data will also be read;
3. **(EX) Execution:** ALU will perform the operation based on the decoded instruction, and produce the result;
4. **(ME) Memory Access:** Sending data to memory or reading data from memory;
5. **(WB) Writing Back:** Write result back to register.

Example 3.1

Describe what happens in the five stages for instruction `ADD X0,X1,X2`.

IF	The binary machine code of `ADD X0,X1,X2` is fectched from instruction memory;
ID	The `opcode` field is used to generate control signals, while registers `X1` and `X2` are read from register file;
EX	ALU receives the two operands, read from `X1` and `X2`, and produce `R[1] + R[2]`;
ME	Since this instruction does not access memory, this stage was passed (but not skipped!);
WD	The result produced by ALU, `R[1] + R[2]`, is written back to register `X0`.

Remarks. One thing we need to notice is, for `ADD` instruction, ME stage was passed, but not skipped. This is true for all instructions that do not need memory access. We defined five stages for all instructions for the purpose of consistency, but they don't necessarily do any stuff in some stages.

3.3.2.1 Stage 1: Instruction Fetching

This is the first stage for an instruction, see Figure 3.21. `PC`, the program counter, sends the address of the instruction to be executed, and the actual instruction will be read from memory and sent out from `Data` on the data bus. We denote the retrieved instruction as `I`.

At the same time, notice `PC` is also sent to an adder. This is to calculate the address of the next instruction, and update `PC`. The adder will add four bytes because each instruction takes four bytes in memory, so adding four will make us move on to the next instruction.

At this time, `PC` will not be updated yet, because the next instruction could be a branch somewhere. The real address of next instruction cannot be decided until EX stage, so we'll talk about it then. For now, what's happening in IF stage can be described as:

```
I      = M[PC]
nextPC = PC + 4
```

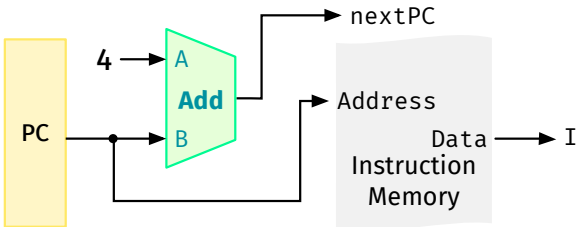


Figure 3.21: Stage 1: instruction fetching.

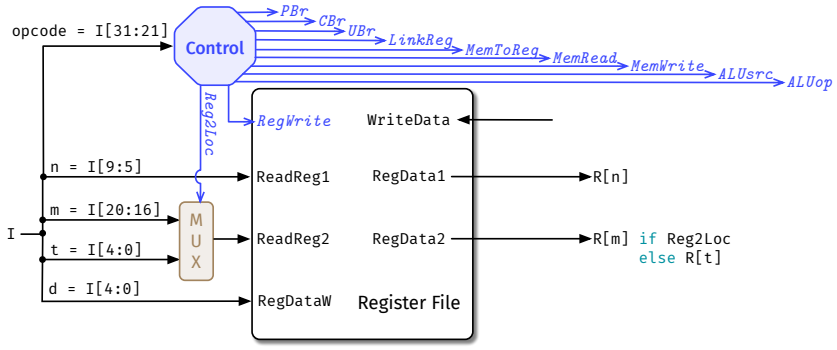


Figure 3.22: Stage 2: decoding. Fields in an instruction is sent to different parts of the register file. The `opcode` is sent to a control unit that can generate control signals.

3.3.2.2 Stage 2: Instruction Decoding

The decoding stage is to take the instruction `I` we just read from instruction memory and send different fields to different data wires, as well as reading register data. We use Figure 3.22 to show some details.

The highest 11 bits are used as `opcode`. This will be sent to the **control unit**, a combinational logic, that converts `opcode` into different control signals used by all parts of the datapath. We will discuss them in the stages when they are actually used.

The register file has two read ports. The first read ports will take `I[9:5]` which encodes the register number, denoted as `n`. This is the same for all instructions. The data read from this register is denoted as `R[n]`.

The second read port needs some work, though. For some instructions such as `ADD Rd,Rn,Rm`, the second read register is `Rm`, encoded in `I[20:16]`. Other instructions, however, such as `LDUR Rt,[Rn,imm]`, the second read register `Rt` is encoded in `I[4:0]`. Thus, before sending the register number to `ReadReg2`, we need a control signal `Reg2Loc` that selects one of them from a multiplexor. When it's zero, it selects `I[20:16]`; otherwise `I[4:0]`.

In summary, this stage can be described as follows:

```
# A group of control signals defined in a 'class'
# All values are initialized as zeros
class sigCntl:
    Reg2Loc,PBr,CBr,UBr,MemToReg,MemRead,MemWrite,ALUsrc =
     ↪  0,0,0,0,0,0,0,0
    ALUop = 0b00

# Obtain opcode from I, and generate control signals
# Control() function returns an object of sigCntl
opcode     = I[31:21]
sigCntl    = Control(opcode)

# Read registers
n, m, t, d = I[9:5], I[20:16], I[4:0], I[4:0]
ReadReg1   = n
ReadReg2   = m if sigCntl.Reg2Loc else t
```

```
16 RegData1   = R[ReadReg1]
17 RegData2   = R[ReadReg2]
```

In the pseudocode above, we first created a class called `sigCntl`, to group all individual control signals together, and initialize them all as zeros. Then after obtaining `opcode`, we use `Control()` to generate an object of `sigCntl` based on the `opcode`, as the control unit is a combinational logic and thus like a function.

We also didn't talk about `WriteReg`, `RegDataW`, and `RegWrite`. Register writing happens in the last stage, not here, so we'll leave it to the end.

3.3.2.3 Stage 3: Execution

This is the main stage where calculation happens. We have to remember that the "calculation" here is not only for arithmetic/logic instructions such as `ADD` and `ORR`. Other instructions also need calculations. For example, when we use `LDUR Rt,[Rn,imm]`, we need to calculate memory address `Rn+imm`. This address calculation is also done by ALU (otherwise where else could it be done?).

As shown in Figure 3.23, there are two sub-jobs done in this stage: Computing and PC updating. We will look at computing first, since that happens before PC updating.

Computing

The main function for ALU is to perform simple calculations, so naturally an ALU takes two inputs, we refer them as `inputA` and `inputB`. `inputA` is simple, which is transferred from `RegData1`. For some instructions, we are not operating on two registers; instead we operate on one register and one immediate, such as `LDUR Rt,[Rn,imm]`. Therefore, `inputB` needs a signal `ALUsrc` to determine which source — `RegData2` or `imm` (encoded in `I[20:10]`) — to use as the second operand. In our description language, it can be done as: `inputB = imm if ALUsrc else RegData2`.

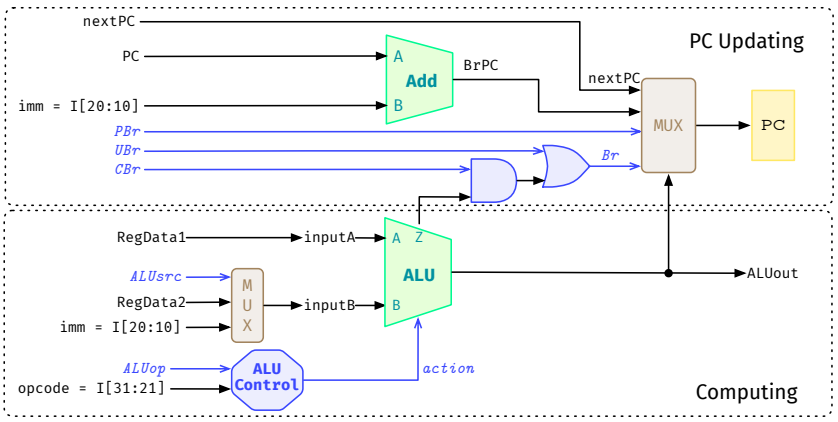


Figure 3.23: Stage 3: execution.

The calculation result from ALU is denoted as `ALUout`. Recall that we mentioned condition codes inside CPU when using instructions such as `CMP`. In our case, we only care about the `Z` flag, so we assume it only produces `Z`.

There are multiple possible calculation, however, such as `ADD`, `ORR`, *etc*. How does it know which one to perform? We would need a control signal called `action`, which is converted based on both `opcode` and `ALUop` through ALU Control. Based on different values of `action`, ALU will perform different calculations and produce corresponding result. We assume only `Z` flag in the condition codes will be generated for simplicity.

We design signal `action` with four bits, and the following eight possible operations for ALU: [6]

6: As to why we use `0000` to indicate and operations and so on, the reason is usually practical, for easier circuit design.

	ALUop	action	ALU operation
AND	10	0000	ALUout = inputA & inputB
ORR	10	0001	ALUout = inputA | inputB
SUB	10	0011	ALUout = inputA - inputB
ADD	10	0010	ALUout = inputA + inputB
LDUR	00		
STUR	00		
B	01	0111	pass inputB, *i.e.*, ALUout = inputB
BL	01		
CBZ	01		
RET	01		
ANDS	11	1000	ALUout = inputA & inputB (set Z)
ADDS	11	1010	ALUout = inputA + inputB (set Z)
SUBS	11	1011	ALUout = inputA - inputB (set Z)

Thus, for this part, we can describe it in the following code:

```
inputA    = RegData1
inputB    = imm if sigCntl.ALUsrc else ReadData2
action    = ALUControl(sigCntl.ALUop, opcode)
ALUout,Z  = ALU(action, inputA, inputB)
```

PC Updating

In Section 3.3.2.1, we learned that after sending current `PC` to the instruction memory, we need to use an adder to add four bytes to `PC` so that we can move on to the next instruction. That is true when all instructions are sequential, but we also need to consider branching instructions.

In general, we have four possible options for updating `PC`: (1) the instruction four bytes after (`nextPC` calculated in IF stage); (2) target instruction by unconditional branch; (3) target by conditional branch; and (4) target instruction pointed by the link register `X30`. So at the rightmost of Figure 3.23, we see there's a multiplexor whose output is sent to `PC`.

This multiplexor receives three inputs:

1. `nextPC`, meaning there's no branch and we'll move to the very next instruction;
2. `BrPC`, the address of target instruction calculated by adding `PC` and immediate together, to perform either conditional or unconditional branch. Recall that in branch instructions such as `CBZ X0,label` and `B label`, `label` is encoded as a PC-relative offset in the binary machine code in the immediate field. Thus, the target address `BrPC` is calculated as: `BrPC = PC + I[21:10]`;
3. `ALUout`, which is used for procedure return. In instruction `RET`, the return address is read from register `X30` as `RegData2`, and passed through ALU as `ALUout`.

Which input is selected to pass through and update `PC` is determined by three control signals: `PBr`, `UBr`, and `CBr`. `PBr` is used for procedure return, *i.e.*, selecting `ALUout` to update `PC`. `UBr` is for unconditional branch, while `CBr` is for conditional. For instructions `B target` and `BL target`, `UBr`'s value is 1 whereas `CBr` is 0. For instructions such as `CBZ`, `UBr` is 0 and `CBr` is 1.

For conditional branches, notice that a signal itself is not enough: to branch or not depends on `Z` flag as well. Thus, we can express the final branch signal `Br` as: `Br = UBr | (CBr & Z)`.

In summary:

```
BrPC  =  PC + imm
Br    =  sigCntl.UBr | (sigCntl.CBr & Z)
PC    =  ALUout if sigCntl.PBr \
                else (BrPC if Br else nextPC)
```

3.3.2.4 Stage 4: Memory Access

Not all instructions will get access to memory, but they still pass this stage. Figure 3.24 shows an illustration of this stage. Note that we mentioned in Section 3.1.5 and Figure 3.15 that there's only one set of data bus, and it can transfer data to and from memory bidirectionally. For clarity reasons, in our datapath, we separate the data bus into `WriteDataM` and `ReadDataM`, two unidirectional buses.

For those who do need memory, such as `LDUR` and `STUR`, we use `MemWrite` and `MemRead` to control if it's reading or writing memory. The memory operation is summarized in the following table:

MemWrite	MemRead	Operation
0	0	No operation
0	1	ReadDataM = M[Address]
1	0	M[Address] = WriteDataM
1	1	Invalid

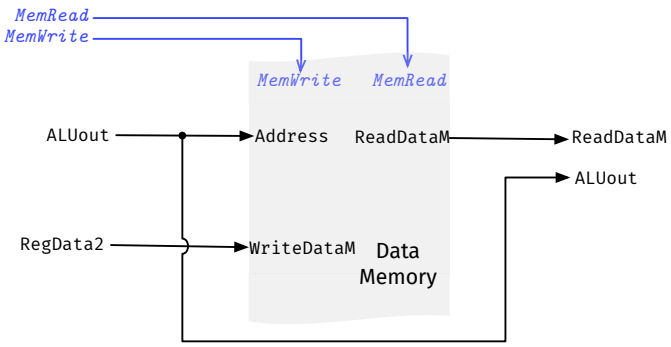


Figure 3.24: Stage 4: memory access. Memory has two control signals `MemWrite` and `MemRead`.

Notice the input `Address` comes from `ALUout`, the output of ALU. This is due to instructions calculating address by adding base register and offset together, *e.g.*, `LDUR Rt,[Rn,imm]`. However, not all ALU outputs represent a memory address, so we branch `ALUout` to bypass the memory as well.

Input `WriteDataM` comes from register read `RegData2`. For example, in `STUR X0,[X1,0]`, `RegData2` is `R[0]`, which will be written to the memory location `ALUout = R[1]+0`. For `LDUR`, `ReadDataM` is the data read from memory, at the address calculated from ALU.

As we see on the right of Figure 3.24, at the end of the MEM stage we have two outputs: one from ALU `ALUout`, and another from data memory `ReadDataM`. Since all we do at this stage is to get access to memory, we will discuss how we handle the two outpus in the next section.

3.3.2.5 Stage 5: Writing Back

The update of register happens in the last stage, as in Figure 3.25. In this stage, we have `RegWrite` to control if we need to write to a register, based on different instructions. Following stage 4, we have `ALUout`, `ReadDataM` and `nextPC`, so we need to choose which one to push into the register as `RegDataW`. We use two signals called `MemToReg` and `LinkReg`, and the last stage can be described as:

```
RegDataW    = nextPC if sigCntl.LinkReg \
                    else ReadDataM if sigCntl.MemToReg \
                                   else ALUout
R[WriteReg] = RegDataW if sigCntl.RegWrite else R[WriteReg]
```

Notice if `RegWrite` is 0, meaning we don't need to write to register, `R[WriteReg]` does not change, hence the last `else` case.

As an example, we see `LDUR X0,[X1]` is to read data from address `R[1]`, and write it to register `X0`. Therefore, `MemToReg` is 1, and `ReadDataM` is selected to write to register. On the other hand, for `ADD X0,X0,X1`, we want to write the addition result of `R[0]+R[1]` back to `X0`, so `MemToReg` is 0, and `ALUout` is written back.

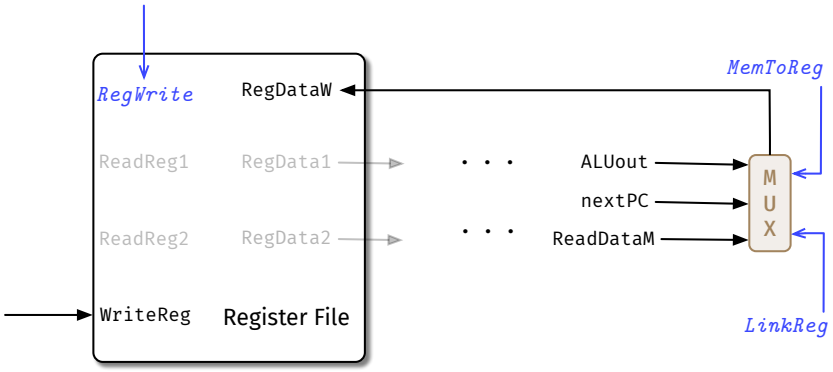


Figure 3.25: Stage 5: writing back.

Signal `LinkReg` is used for instruction `BL` specifically, where we need to write the return address, *i.e.*, `PC+4`, back to register `X30`.

Now that the last stage has finished, and PC has been updated in stage 3, we are ready to start the next instruction and walk through the five stages again.

3.3.2.6 Summary

We use Figure 3.26 to summarize the five stages, where the datapath diagram and dscription language are side by side.

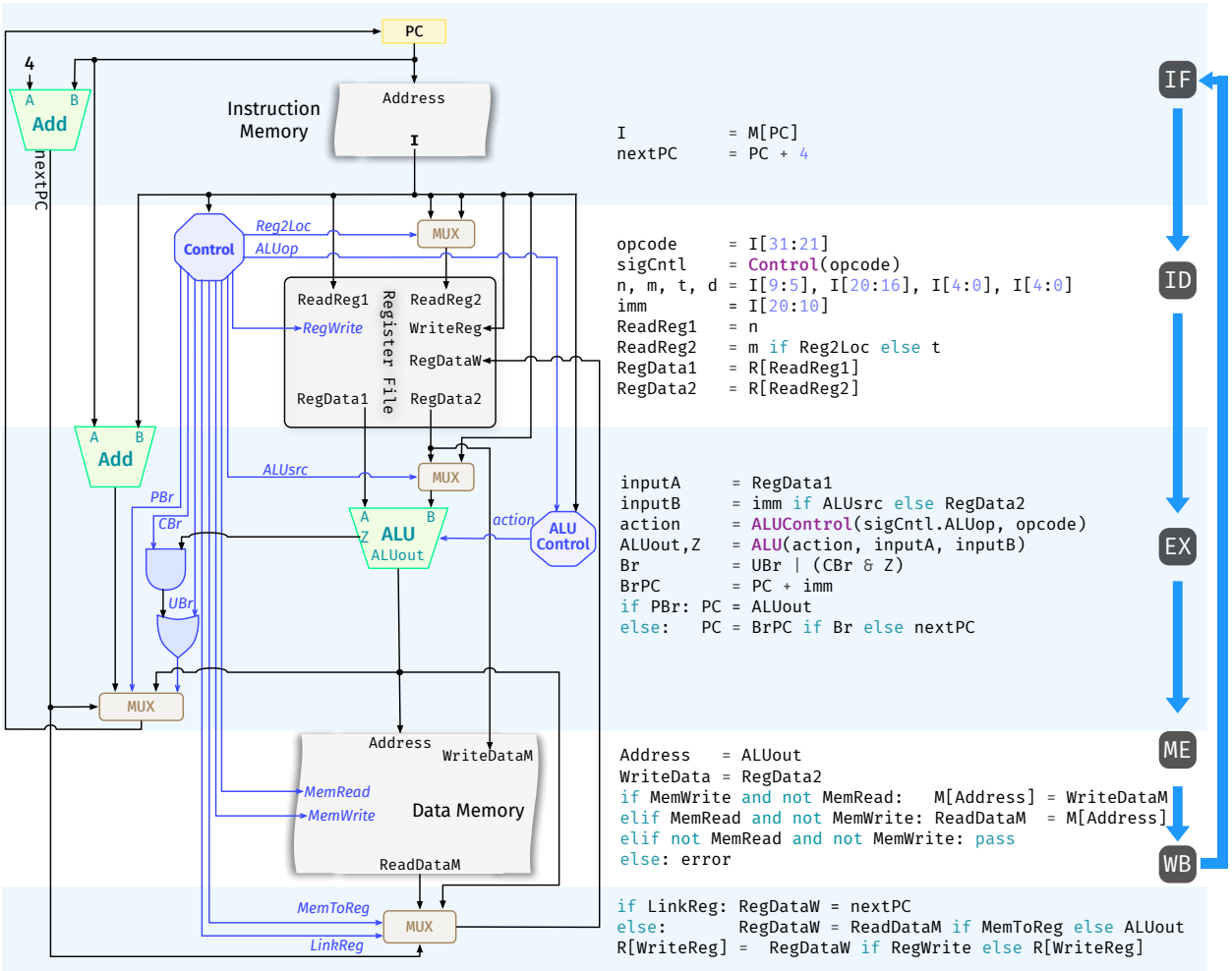


Figure 3.26: A summary of five stages each instruction goes through in the datapath, with description language on the side.

3.4 A Pipelined Datapath

The sequential implementation is a simple model and shows important ideas and concepts in building a datapath. It is a very inefficient one, because when an instruction goes through later stages such as MEM, all the other components are idle while the following instruction is waiting, which is a waste of resources. In this section, we'll modify our model to a pipeline to address this problem.

3.4.1 Operating a Pipeline

Before we measure the efficiency of a pipeline, we'll define some terminologies.

Throughput: We express throughput in units of giga-instructions per second (abbreviated GIPS), or billions of instructions per second;

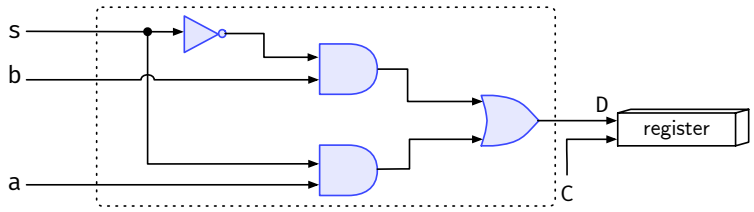


Figure 3.27: An example of combinational logic, where the input is three-bit, and the output `D` is one bit. When the clock rises, the output `D` is written to the register.

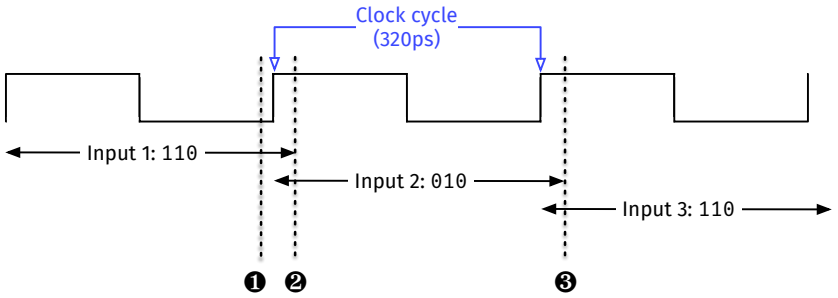


Figure 3.28: A sequence of three inputs: `110`, `010`, `110` with an unpipelined version.

Latency: The total time required to perform a single instruction from beginning to end is known as the latency; Latency is the reciprocal of throughput.

To fully understand pipeline in a datapath, let's start with a very simple example, where we simply send some input data to a combinational logic that converts it into output data, and we store it to a register. See Figure 3.27.

In Figure 3.27, the input has three bits: `sba`, and the component surrounded by dotted lines is the combinational logic, whose output is `D`. Recall from Section 3.1.3.3, the register is also controlled by a clock signal `C` — the data will be written into register *only* when there's rising edge of the clock signal.

Assume it takes $300ps$ to get `D` from `sba`, which is the delay of the combinational logic, and it takes $20ps$ to push the data into the register. In Figure 3.28, we show a sequence of three inputs: `110`, `010`, `110` with an unpipelined version. A **clock cycle** is the period between two consecutive rising edges. In order to process one instruction at a time in every clock cycle, we have to make sure we leave sufficient time for the delay caused by the combinational logic, so each clock cycle takes $300 + 20 = 320ps$. To get output from the three inputs, we need to wait $320ps \times 3 = 960ps$.

However, if we look at the combinational logic closely, we'll notice that while the first input passed through the two `and` gates, the `not` gate is idle. Since the second input will use this `not` gate anyway, why don't we send input 2 there already? To clearly show this idea, as in Figure 3.29, we separate the combinational circuits into three stages, where each stage's input is previous one's output. One input 1 passed through stage 1 and is currectly in stage 2, we can send input 2 to stage 1, and so on. Now you can see at the peak of the execution, we will have three input signals running through the combinational circuits at the same time. This is usually called a **three-way pipeline**.

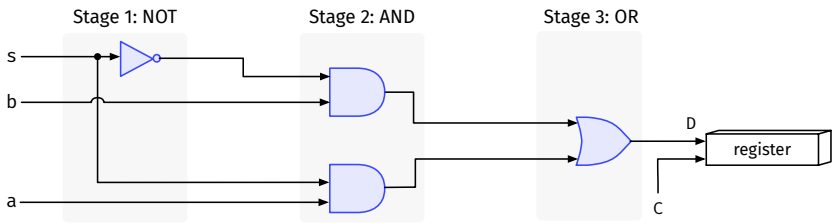


Figure 3.29: A three-way pipeline structure where each stage runs one input.

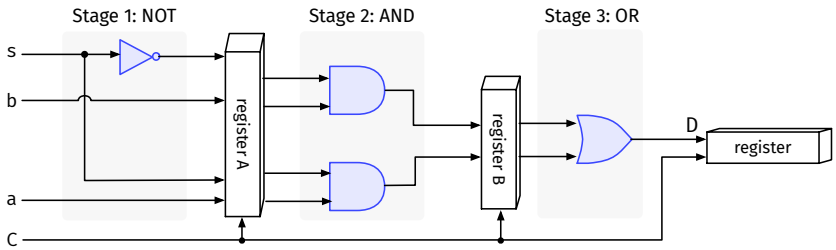


Figure 3.30: We added two registers between the three stages as "barriers", to make sure the signals in each stage will not be interrupted or overwritten.

3.4.1.1 Pipeline Registers

The idea is simple and straightforward, but there are problems. When input 1 moves to stage 2, we were supposed to send input 2 to stage 1. However, notice only signal s will go through stage 1; signals a and b will shoot the and gate right away, but currently it's used by input 1. Because combinational logic will respond to any signal input change immediately, now in fact input 2 is in both stage 1 and 2, while the data from input 1 has been overwritten and lost.

Therefore, we need to put a "barrier" between the stages, as if telling the signals: "hold on a second ✋; let's make sure previous signals have finished the job!" This "barrier" can be easily implemented using registers.

As shown in Figure 3.30, we inserted two registers, A and B, between the three stages, called **pipeline registers**. Register A stores 4 bits of data, while B 2 bits. When there's a rising edge of the clock, they'll store the output from the previous stage, and then use them as the input for the next stage. Because registers only write data on rising edge of the clock, even if previous stage has changed its signals, during the cycle it won't overwrite the existing data in the registers.

Now that we have separated the combinational circuits into three stages, each of the stage has a shorter delay, so we assume each stage takes $100ps$. Since we added two registers, we also need to consider the delay of writing to registers, and we can assume the delay is the same for all registers, which is $20ps$.

We show the timeline in Figure 3.31 using this 3-way pipeline, where each clock cycle executes one stage. Each cycle takes $120ps$ ($100ps$ for combinational circuits, and $20ps$ for writing registers), so in total for completing the three inputs the delay is $360ps$. Comparing to the unpipelined version in Figure 3.28 which takes $960ps$, the total delay has been greatly reduced.

One thing we do need to notice is that in the pipelined version, each individual instruction's latency has been increased actually, from $320ps$ to

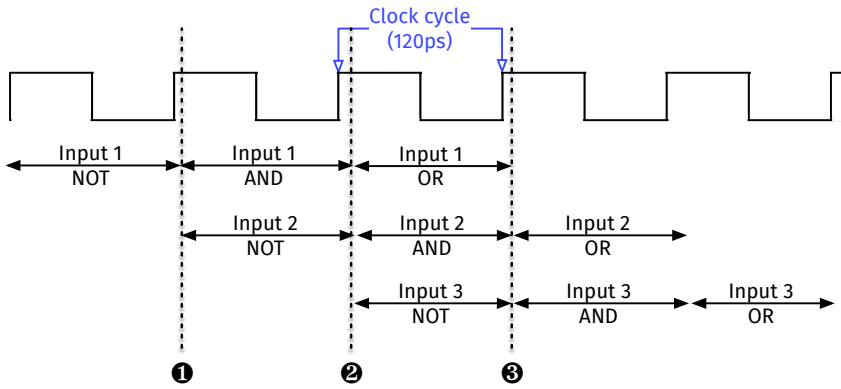


Figure 3.31: When we separate the combinational logic into three stages, the clock cycle can be shorten to only execute one stage. At the peak of the system, between time 2 and 3, all logic gates are working on different instructions, which greatly improves the throughput and reduces resource waste.

$120ps \times 3 = 360ps$. Regardless, pipeline is still a more efficient model because in total the latency is almost only one third of the unpipelined model.

Quick Check 3.3

True or false, and why?

1. By pipelining the CPU datapath, each instruction will execute faster, resulting in a speed-up in performance;
2. A pipelined CPU datapath results in instructions being executed with higher latency and higher throughput.

3.4.1.2 From Sequential to Pipeline

In the previous example, the analogy to our sequential datapath is obvious. Our datapath can be treated entirely as a combinational logic separated into smaller parts, and only the last stage — writing back — is sequential.

To apply pipeline to the datapath of five stages, we use Figure 3.32 to show this idea. In the figure, we have a sequence of three instructions to execute. Using the sequential model from previous section, one instruction takes over the entire sequence (clock cycle), so the second instruction cannot start executing until its previous instruction has finished all stages.

If we "stack" them, as at the bottom of Figure 3.32, we see that it takes much shorter time to complete the entire sequence. For example, at time ①, `LDUR` is in ID stage, and at this time `PC` is idle, so why don't we start putting `ADD` instruction in IF stage? Now at time ①, we have two instructions running at the same time.

Now that the big picture of constructing a pipeline is clear, we're going into details on the datapath in the next section.

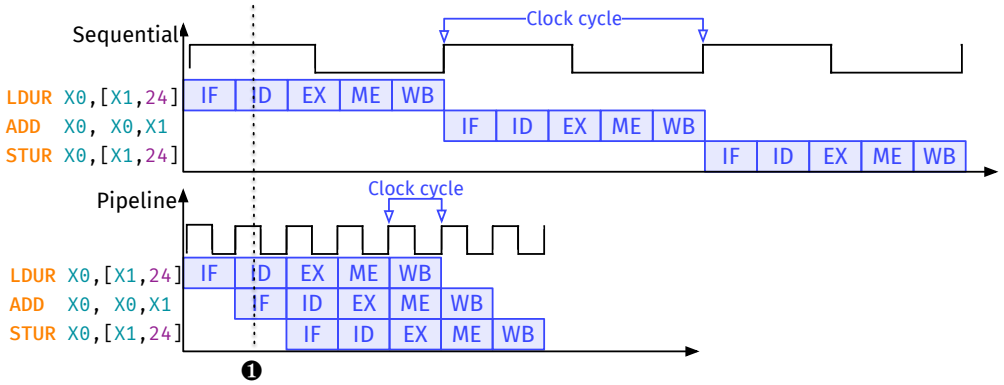


Figure 3.32: The horizontal axis is a time line. At the top we run one instruction through all five stages at a time, so it takes much longer to complete all three instructions. At the bottom, we run multiple instructions at the same time, which resembles a pipeline.

3.4.2 Adding Pipeline Registers to Datapath

Similar to the simply circuit example, we need to add pipeline registers between every pair of stages (except WB and IF). The pipeline registers store different values, depending on the data that each stage needs to store. We name them using the names of the stages they insert: IFID, IDEX, EXME, and MEWB. In Figure 3.33, we add these four pipeline registers, where we also show the data stored in them.

Control signals also need to be pipelined, but not all signals will be used in every stage. Based on the stages they are used, we group them into EX, ME, and WB. EX contains `PBr`, `CBr`, `UBr`, `ALUsrc`, and `ALUop`. ME and WB signals are not used in the EX stage, so we pass them on to the next pipeline register. ME contains `MemWrite` and `MemRead`, while WB `LinkReg`, `MemToReg`, and `RegWrite`.

We will use similar description language in the following sections. For example, `IDEX.nextPC` indicates the data `nextPC` currently in register `IDEX`; `EXME.MemRead` is the signal `MemRead` in `EXME`; and so on.

We will also use a *description diagram* in examples to show the instructions and stages in each cycle, much similar to Figure 3.32. Figure 3.34 shows a pipeline diagram of a sequence of instructions. We mark the cycle number at the top from 1 to 9, and each cycle can have at most five instructions running at different stages at the same time, since our datapath has five stages.

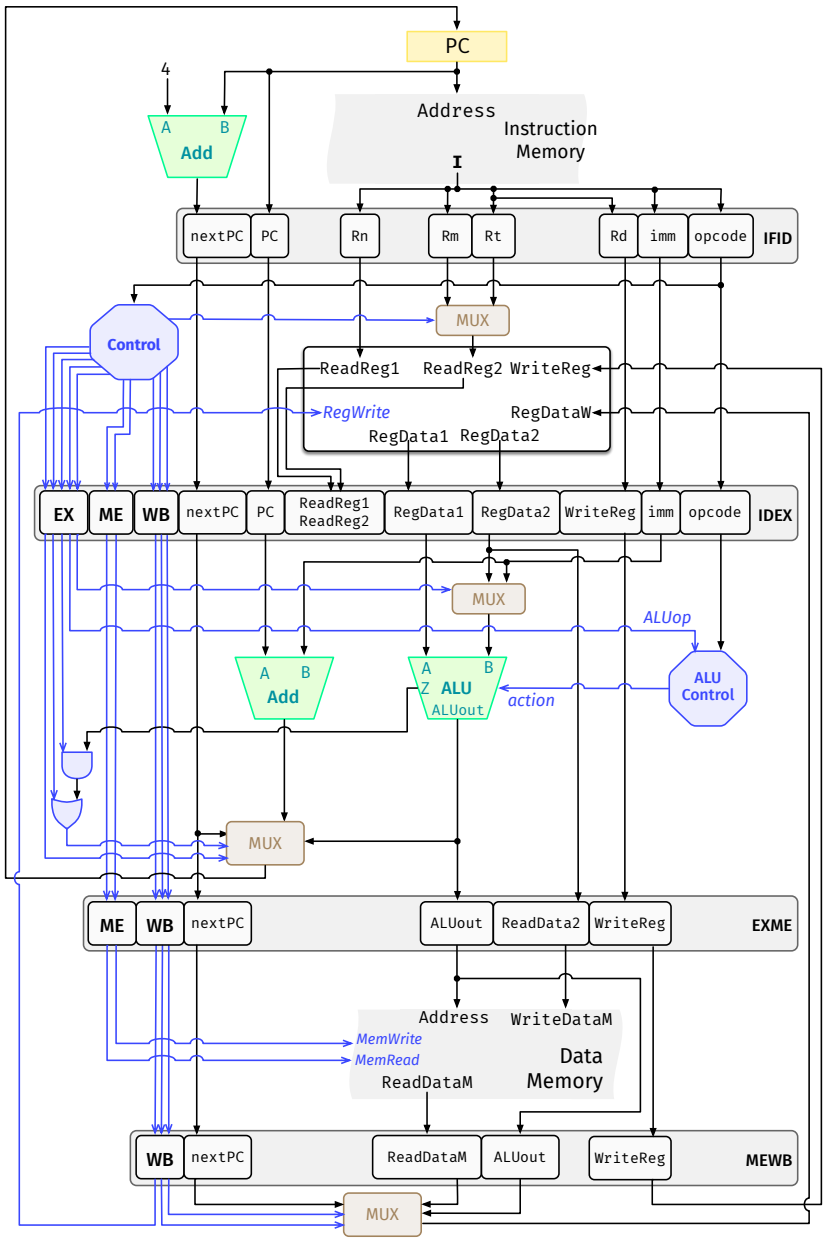


Figure 3.33: A detailed datapath with pipeline registers.

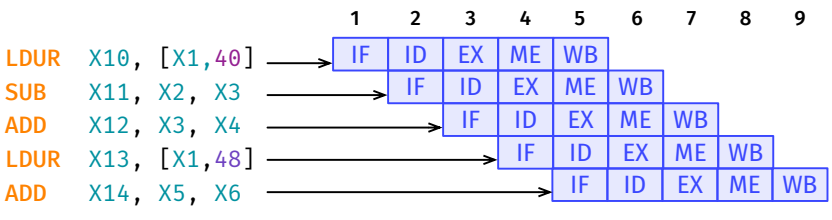


Figure 3.34: A pipeline diagram showing the progression of instruction executions.

3.5 Hazards

Pipeline is efficient, but if we just pay a little bit attention to some details we'll realize it can lead to wrong executions. Based on the error it can lead to, we categorize it to two types: **data hazard** and **control hazard**, and we'll discuss them in detail in this section.

3.5.1 Data Hazard

Data hazard refers to wrong values in the pipeline are read and written. Let's look at a simple example in Figure 3.35.

We assume in the beginning, we have the following register values:

X0	X1	X2	X3	X11	X12
50	10	10	20	-125	180

At cycle 3, instruction 1 `SUB` has read register values from `X1` and `X3`, so at the end of EX stage, `ALUout` is the result of `R[1]-R[3] = -10`, which will also be written to `X2` at the end of WB stage. As seen in Figure 3.35, the result `-10` is carried over through ME and WB, and `X2` is only updated at the end of cycle 5. Before that, `X2`'s value is always 10, the old value.

Now this is the key point of data hazard. At cycle 4, instruction 3 `ADD` which uses `X2` as one of the source operands, is already in the ID stage where the *old value* of `X2` is read. In other words, the new value hasn't even been updated back yet. This wrong value is carried to EX stage, and surely the calculation result is wrong. It's supposed to be 40, but now we see it's 60.

Had we use the sequential model, this problem would've been avoided, because in sequential model an instruction starts fetching and reading registers only after previous instructions have written back registers.

3.5.1.1 Stalling the Pipeline Manually

Our first solution is the most straightforward one—stall the pipeline. Recall that the problem we have is the instruction that needs to read an updated register starts executing too fast. Then why not let it wait one cycle or two? That's exactly how stalling works.

Following the example above, let's see how we can *manually* implement stalling. Simply put, what we want is to delay the execution of the instruction `ADD`, but we have two questions. First, delay to which cycle? As in Figure 3.35, we see that in cycle 5 has `SUB` instruction finished updating `X2`. Because reading registers is a combinational circuits—the read port will read the updated value as soon as the register value has changed, as long as we let instruction 3's ID stage align with instruction 1's WB stage, we're good.

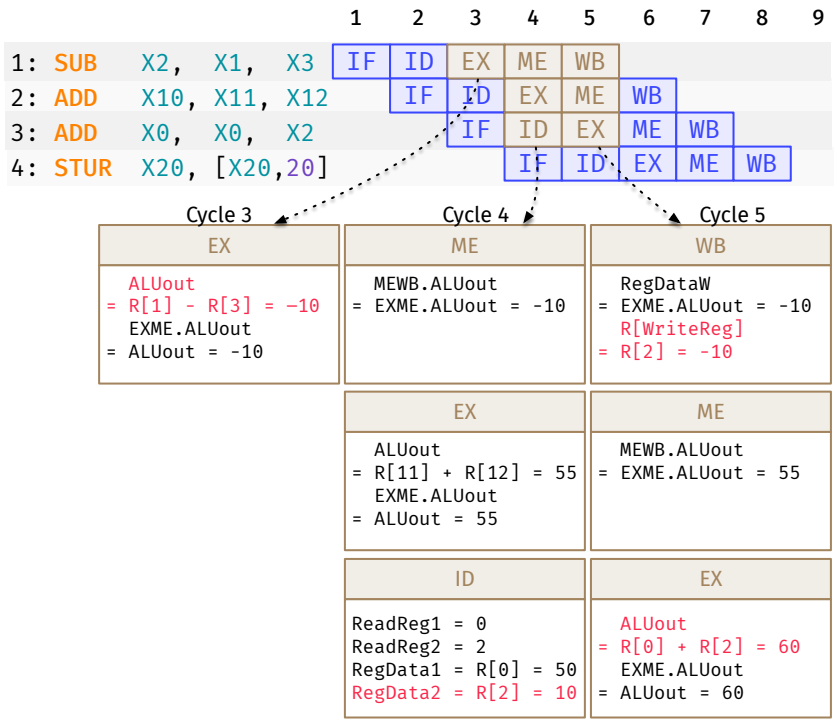


Figure 3.35: A sequence that can lead to data hazard, due to dependencies between instructions. Register X2 in the first instruction is the destination, but also one of the source operands in the third instruction. At cycle 4, instruction 3 has already read X2's old value, but it hasn't been updated from instruction 1 yet.

Second question is, the pipeline constantly fetches instructions from PC, so we need to add some instruction between instructions 1 and 3. It cannot, however, do anything other than making the stages busy, and cannot interfere with the logic of the program.

Solution 1: Rearrange instructions. This is a useful trick, but not applicable for all programs. In terms of the program in the above example, if we swap instruction 3 and 4, then the problem is solved. See Figure 3.36.

Again, in some cases, this solution cannot be used. For example, for a sequence such as the following:

```
1 ADD   X2, X2, X0
2 SUB   X2, X2, SP
3 LDUR  X0, [X2,48]
```

this trick doesn't work anymore, because we see data dependency between every pair of instructions; no other instructions can be swapped.

Solution 2: Inserting `NOP` instruction as necessary. `NOP` means "**no op**eration"; its only purpose is to "waste" some cycles so that some instructions can be delayed. This instruction does nothing but taking a place in the pipeline stages.

In Figure 3.37 we added one `NOP` between the two `ADD` instructions. We only need one `NOP` inserted in this example, because as long as `ADD`'s ID stage is aligned with `SUB`'s WB stage, we're all good. More `NOP`s only waste cycles unnecessarily. With this, we can see that the position of `NOP` is not

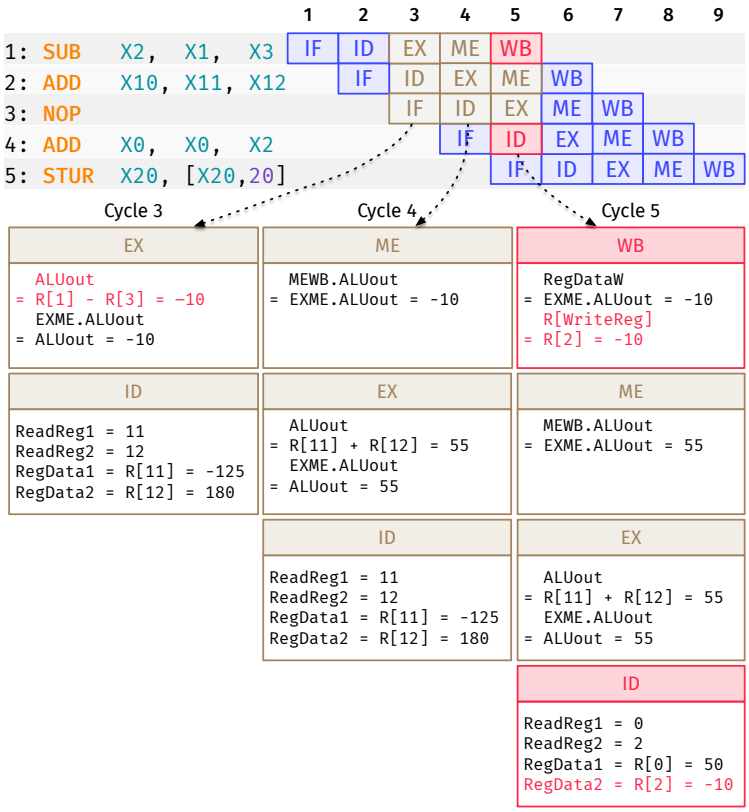


Figure 3.36: Because there's no dependency on X2 in instruction STUR, we swap it with ADD to align its ID stage with SUB's WB stage.

unique—we can certainly add one NOP right after SUB instruction, and it'll have the same effect on delaying the pipeline.

One thing we'd like to mention about NOP. In the detailed boxes in Figure 3.37, notice in cycles 4 and 5, NOP's ID and EX stages are exactly like the ones in its previous ADD instruction. This is because, like we said, NOP instruction does not do anything in the pipeline—meaning it doesn't change or alter what's already there. When its previous instruction moves along the pipeline to the next stage, the data/control signals stay there unless its next instruction moves in and overwrites them. NOP doesn't do anything, so the data will not be changed.

Remarks. Note that the two solutions presented above do not need to be exclusive. Sometimes re-arranging the instruction sequence is not enough, and so we'd have to insert NOPs as well.

Quick Check 3.4

(Fall 21 Midterm 2) Rearrange the following instructions to avoid data hazard. You can insert NOPs as needed.

```
1 LDUR X1, [X0, 24]
2 ADD  X2, X1, X0
3 SUB  SP, SP, 16
4 BL   proc
```

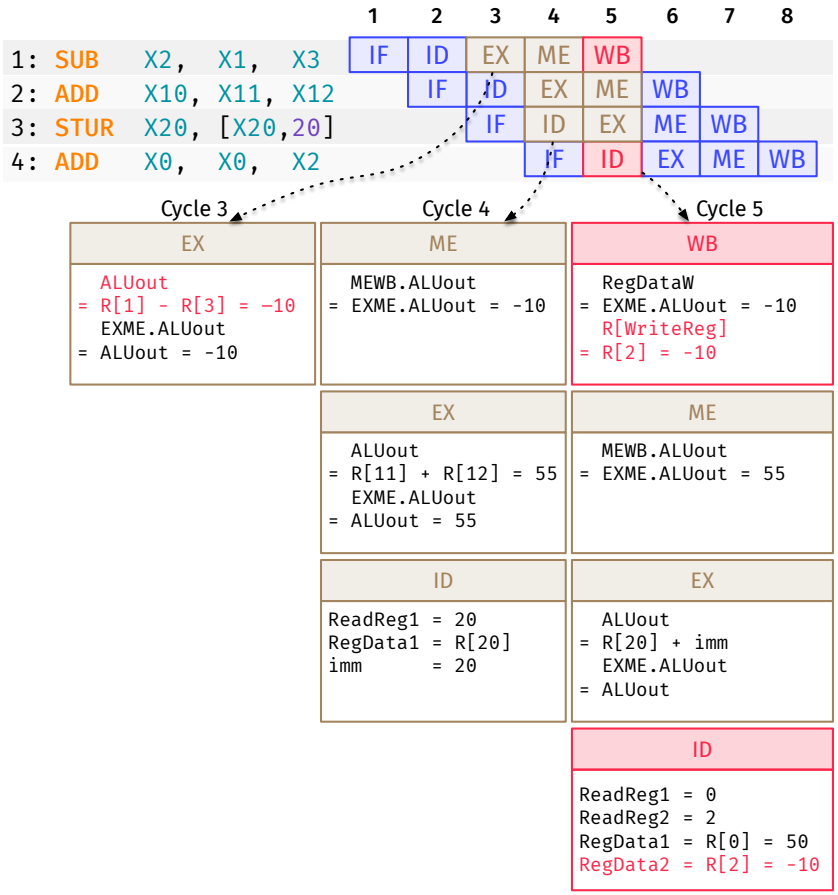


Figure 3.37: Inserting `NOP` instruction allows us to delay instructions that depend on the completion of earlier instructions. In this example, to make `SUB`'s WB and `ADD`'s ID stages align, we only need to add one `NOP`. Certainly in some cases more `NOP`s may be needed.

> Note: explain why you rearrange the instructions the way you did. Without explanation you'll only get half of the points.

3.5.1.2 Stalling the Pipeline Automatically

It'll be really exhausting if resolving data hazards relies on programmers entirely—the programmers should focus on the task itself, and let the machine resolve these hazards. Therefore, we can create a combinational circuits called **hazard detection unit** to automatically detect data dependencies between stages, and stall instructions as necessary.

Let's consider the example presented in Figure 3.38. At cycle 4, instruction 1 is at ME stage while instruction 3 at ID stage. There's data dependency between them, so we need to make sure instruction 3 reads the correct value of `X1`. After detecting this hazard, instruction 3 has been **stalled** in the ID stage in the next cycle. Because at cycle 5, instruction 3 is still at ID stage, its following instruction `STUR` is held in IF stage as well. Our pipeline has five stages and they should all be occupied. If instruction 3 cannot move to EX stage, what should we put in there? The processor will automatically insert a "bubble", which is basically a `NOP` instruction, and it'll pass EX, ME, and WB stages in the following cycles.

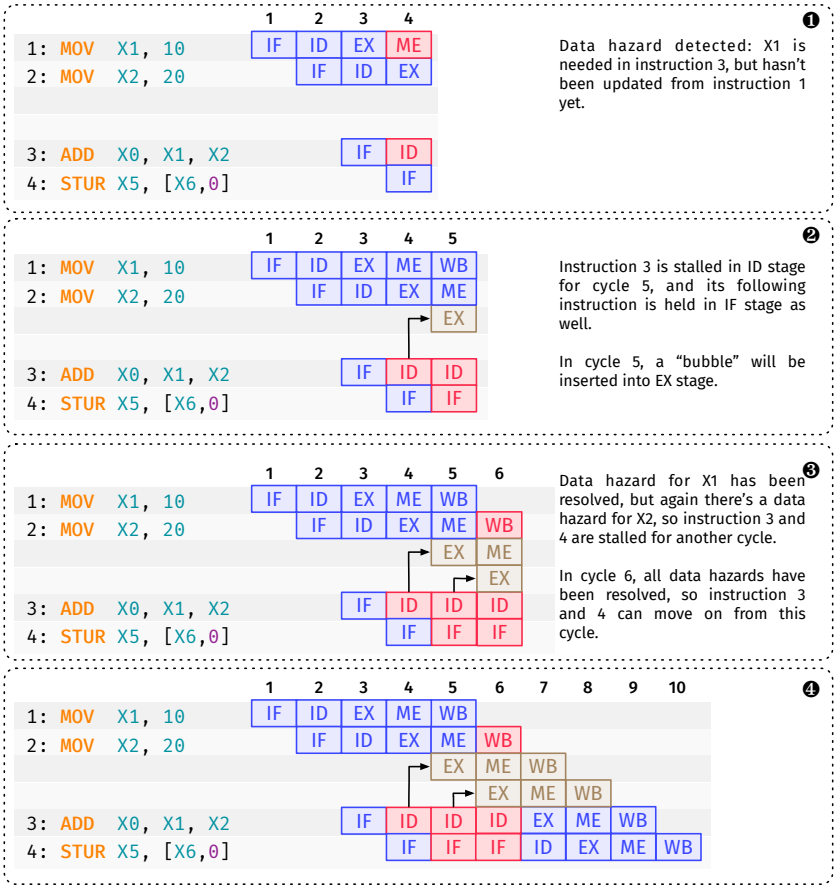


Figure 3.38: As long as there's a data hazard, we'd postpone the instruction and its following instructions by **stalling** them in place, and let `NOP`s move along the pipeline. Once all data hazards have been resolved, stalled instructions can restart.

Inserting one bubble is not enough, though, because we notice there's another data hazard for `X2` at cycle 5. In this case, we keep stalling instructions 3 and 4, and insert another bubble. At cycle 6, both `X1` and `X2` have been updated, so instruction 3, which is currently held at ID stage, can read both `X1` and `X2` successfully. Thus, from cycle 7, we move instructions 3 and 4 along the pipeline, and the data hazard has been resolved.

Now let's analyze in what situation will there be data hazard. From previous examples and Figure 3.38, we have seen that typically, if one instruction's trying to read a register that hasn't been updated from previous instructions, there's data hazards. In other words, the instruction at ID stage relies on its previous instructions whose `RegWrite` control signal is 1, and yet they haven't reached to the WB stage. Such instructions are either in EX or ME stages. Using description language, we can have the following condition to check data hazards:

```
if IDEX.RegWrite:
    if  IDEX.WriteReg == IFID.ReadReg1 or \
        IDEX.WriteReg == IFID.ReadReg2:
        # Stall the pipeline
if EXME.RegWrite:
    if  EXME.WriteReg == IFID.ReadReg1 or \
        EXME.WriteReg == IFID.ReadReg2:
```

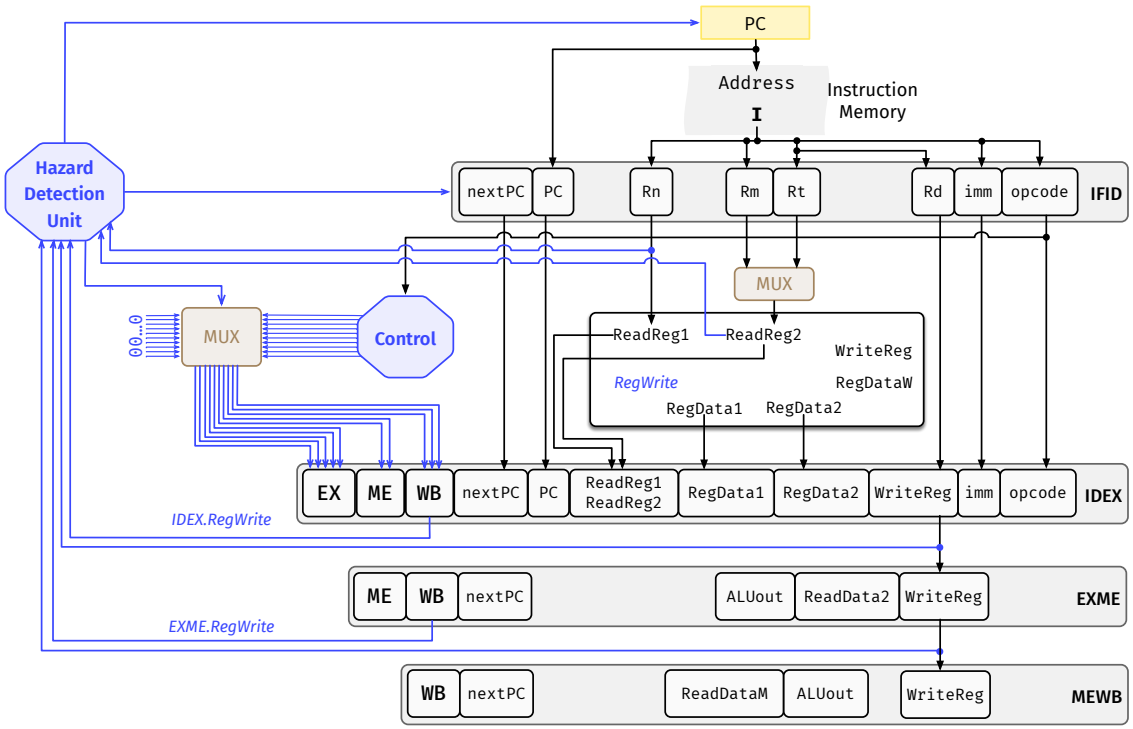


Figure 3.39: Hazard detection unit receives signals from multiple stages (representing multiple instructions), and stall instructions currently at ID and IF stages, and insert bubble to EX stage.

```
8         # Stall the pipeline
```

With this pseudocode, implementing a hazard detection unit is very straightforward—we just need to create a combinational circuits that can stall the pipeline. In Figure 3.39, we added a hazard detection unit, where it receives signals from multiple stages.

Once data hazard detected, this hazard detection unit does two jobs:

1. Stall instructions currently at ID and IF stages. If data hazard has been detected, it'll overwrite PC and the values in `IFID` register. The instruction at IF stage will be fetched again, since PC will not be updated. The instruction at ID stage will stop updating `IFID` register's values from next instruction, and will stay the same. Thus, both instructions have been held without moving along the pipeline;
2. Insert bubble to EX stage. Before the EX stage starts, EX, ME, and WB signals in `IDEX` reprensted the instruction that we want to stall. If there's data hazard, we cannot let them move to the next stage. Therefore, instead of passing control signals generated by the control unit to `IDEX` directly, we send them to a multiplexor, and let the hazard detection unit decide if those signals can be passed to `IDEX`. If there's data hazard, the multiplexor will pass all zero signals to `IDEX` instead, which is used as `NOP` instruction.

3.5.1.3 Forwarding Data

Stalling a pipeline can surely avoid data hazards, but it also slows down the pipeline. In many cases, forwarding data right from EX stage back is faster. Thus, we create another combinational circuits called **forwarding unit**, as shown in Figure 3.40. The main function of this unit is to detect if there's a data hazard. If there is, it'll overwrite the data read from the registers. This way, a new value will be used as the input operand(s) of ALU, instead of the old and not updated values from register read.

For the two operands from ALU, we use a four-way multiplexor to select one of the three possible inputs: one from the ALU result of its previous instruction, `EXME.ALUout`; one from the written back value from `RegDataW`; one from register read, either `RegData1` or `RegData2`.

Since it's a four-way multiplexor (see Section 3.1.2.2), we'd need two bits of signals for selection, which are also the output of the forwarding unit. We call them `FA` and `FB`, used for operands A and B of ALU. Thus, we can establish a truth table for the multiplexors as follows:

	Value	Forwarding source	Explanation
FA	00	`IDEX.RegData1`	A is from the register file
	01	`EXME.ALUout`	A is forwarded from previous ALU result
	10	`RegDataW`	A is forwarded from `RegDataW`
FB	00	`IDEX.RegData2`	B is from the register file
	01	`EXME.ALUout`	B is forwarded from previous ALU result
	10	`RegDataW`	B is forwarded from `RegDataW`

The description language for the multiplexor can be written this way:

```
if   FA == 0b00: inputA = RegData1
elif FA == 0b01: inputA = EXME.ALUout
else: inputA = WriteReg
```

which is the same for operand B as well.

Here are some details we still need to work on:

A. Conditions for Forwarding

Now that the output of the forwarding unit and their situations have been identified, next step is to consider when to forward, and design corresponding conditions for the unit. Let's focus on the case where `FA==01` first, *i.e.*, the operands are forwarded from previous ALU result. If ALU result of instruction `i` will be used as one of the operands for the following instruction `i+1`, apparently `i`'s `WriteReg` matches one of the operands of `i+1`. Therefore, we have [7]

```
if  EXME.RegWrite and \
    EXME.WriteReg == IDEX.ReadReg1 and \
    EXME.WriteReg != 31:
    FA = 0b01
```

7: We only write conditions for operand A, and those for operand B can be easily derived.

Notice in the condition, we added `EXME.WriteReg != 31`, because `X31` is `XZR` and it will never be overwritten, and thus there's no data hazard involved with this register.
The second possible condition is `FA==10`, meaning operand A is forwarded from `RegDataW`, which is one of the three signals—`MEWB.nextPC`, `MEWB.ReadDataM` from memory read, and `MEWB.ALUout` from previous ALU result. Following the code above, we can keep checking the conditions as follows:

```
elif MEWB.RegWrite and \
     MEWB.WriteReg != 31 and \
     MEWB.WriteReg == IDEX.ReadReg1:
    FA = 0b10
else:
    FA = 0b00
```

B. Multiple Possible Forwarding

Notice in the code listing above we put the checking from ME stage in `elif` instead of an independent `if` statement. It means that if we can receive forwarded data from EX stage, we will **not** consider the possibility from ME stage anymore.
Look at the following example:

```
ADD  X1, X1, X2
ADD  X1, X1, X3
ADD  X1, X1, X4
```

When the third `ADD` instruction is at ID stage, we have new `X1` values forwarded from both EX and ME stages, corresponding to the result from the second and the first instructions. So which one should we use? Apparently we should use the newest one, *i.e.*, calculated from the second `ADD`. By using `elif` for ME stage condition, we successfully avoided this problem.

C. Removing Wires for Stalling

After added forwarding unit, we learned that this is a faster option, because it doesn't waste cycles and slow down our pipeline. However, there's one situation that forwarding doesn't work. Let's look at the following sequence:

```
LDUR   X0, [X1, 48]
ADD    X2,  X0, X0
```

where the destination register of `LDUR` is exactly both operands for the next instruction. When `ADD` instruction is at ID stage and needs to read the correct value of `X0`, `LDUR` is still reading the memory, and will **not** extract the correct value of `X0` until next cycle. However, after `LDUR` reads the correct value and moves to WB stage, `ADD` has moved to EX stage, which uses the wrong value of `X0`. Forwarding

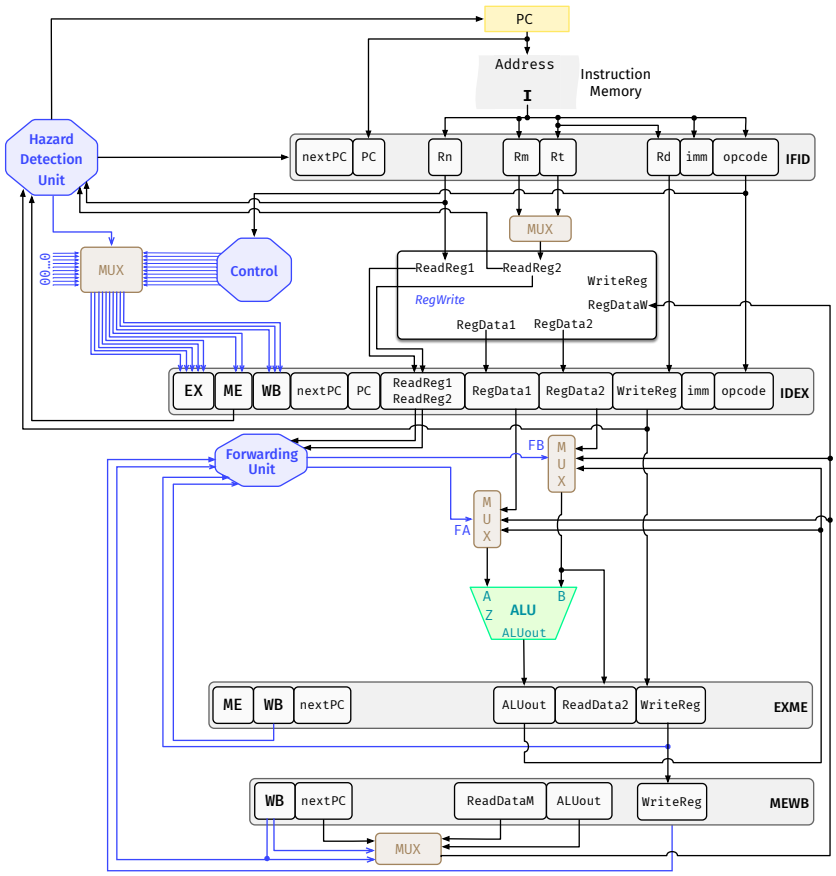


Figure 3.40: The forwarding unit will take signals from ME and WB stages, and overwrite ALU operands.

in this case fails, and apparently the only solution is to insert a bubble between the two instructions.

As shown in Figure 3.40, we removed some of the wires for hazard detection unit, because forwarding unit has been added and its more efficient. We did save one case for it, where it receives `IDEX.MemRead` and `WriteReg`, just to deal with the situation we discussed above. Thus, we modify the description language in Section 3.5.1.2:

```
if IDEX.MemRead:
    if IDEX.WriteReg == IFID.ReadReg1 or \
       IDEX.WriteReg == IFID.ReadReg2:
        # Stall the pipeline
```

3.5.2 Control Hazard

Control hazard refers to the wrong execution of programs (not control signals), which is mainly due to branching. Consider the code sequence:

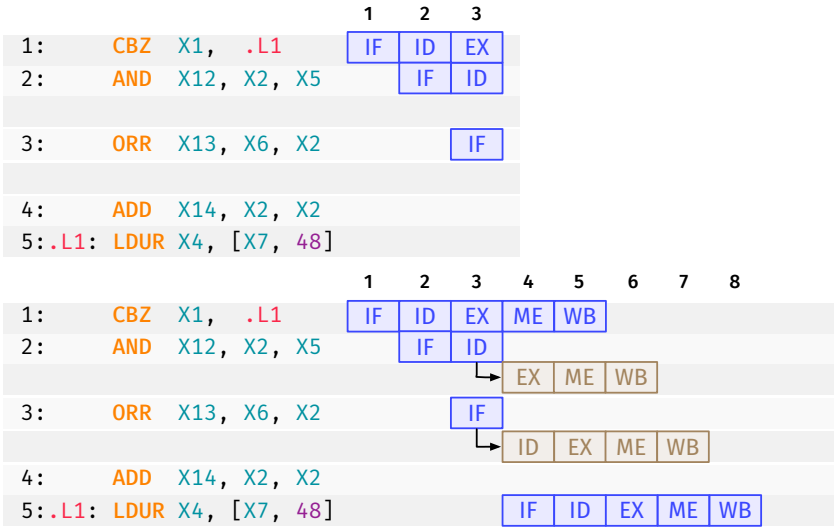


Figure 3.41: At cycle 3, the value of `X1` has been determined. Assume it is zero, then we need to branch to `.L1`. The two instructions `AND` and `ORR` that are already in the pipeline will not continue executing; instead, we flush them and let `NOP`s move along the pipeline.

```
1       CBZ  X1,  .L1
2       AND  X12, X2, X5
3       ORR  X13, X6, X2
4       ADD  X14, X2, X2
5  .L1: LDUR X4, [X7, 48]
```

If `X1` is not zero, everything is fine; but if it is zero, after execution of `CBZ` we should execute `LDUR`. Now put this into the pipeline, we'll see the problem. The value of `X1` can only be determined at EX stage. [8] At this time, the instructions `AND` and `ORR` has already been put into the pipeline. If `X1` *is* zero, we have two instructions in the pipeline that are not supposed to be there.

8: `X1` is read from ID stage, but if it's zero or not cannot be detected until at EX stage ALU receives `X1` and produces zero flag.

A simple solution is just to cancel the wrong instructions in the pipeline when we notice there's a control hazard. This operation of "cancelling" is called **flush**. This is pretty much like inserting bubbles and `NOP`s, which prevents wrong instructions from moving on to next stages in the pipeline. See Figure 3.41.

3.5.2.1 Branch Prediction

The inserting-bubble method we used above seems working fine; it wastes only two cycles and it's only 50% of the chance. Notice, however, that the reason why we only waste two cycles is because we have five stages, and the value of a register can be confirmed as early as the third stage. Modern processors have longer and deeper pipelines, *i.e.*, more stages. If a processor has N stages, and the value of a register can be confirmed at the $\frac{N}{2}$ stage (in the middle), then we'll have to waste $\frac{N}{2}$ cycles. For a large N, this apparently is not ideal at all. In other words, the penalty for control hazard becomes untolerable as the pipeline deepens. Therefore, we introduce two common practice for modern processors.

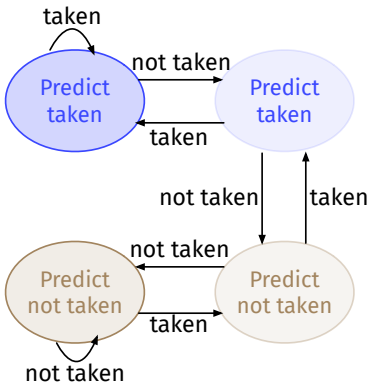


Figure 3.42: Two-bit predictor.

Static Prediction

Static prediction is based on the characteristics of the programs. For example, at the end of a loop body, we always predict that it'll branch back, because a loop typically runs multiple iterations. If the branch is not part of a loop but to a procedure or other part of the code, we always predict that it'll not take the branch. In other words:

- Predict branch taken if it's going backwards;
- Predict branch not taken if it's going forwards.

Dynamic Prediction

Dynamic prediction uses a hardware to record the behavior of every branch — if it's taken or not — and make decisions based on current status.

For example, two-bit predictor will record the branch behavior. As in Figure 3.42, it predicts the branch based on its past two predictions and the actual behaviors.

As we see, both prediction strategies can only reduce the penalties. If a misprediction has happened, we still need to insert bubbles and re-fetch the correct target instruction. In fact, misprediction is almost non-avoidable. We will not cover more on this topic, so if you're interested, feel free to explore!

3.6 Performance Evaluation

Throughout this chapter, we have seen that in CPU design it is important to control the period of each clock. Each clock should be long enough to accommodate all stages, but also not so long to have unnecessary delays. Thus, in performance evaluation, we first care about **clock period**, which is the time spent on each clock cycle. We typically use **picoseconds** (abbreviated "ps"), or 10^{-12} seconds to denote clock period. For example, $250ps = 0.25ns = 250 \times 10^{-12}s$.

What we are most familiar with, especially when buying a computer, is the *reciprocal* of clock period, called **clock rate** or **clock frequency**:

$$\text{clock rate} = \frac{1}{\text{clock period}}. \tag{3.2}$$

This measurement denotes how many cycle a CPU can run each second, and its unit is Hz. For example, for a computer with clock period of $250ps = 2.5 \times 10^{-10}s$, its clock rate is

$$\begin{aligned} \text{clock rate} = \frac{1}{2.5 \times 10^{-10}s} &= 4 \times 10^{9}\text{Hz} && (3.3)\\ &= 4 \times 10^{3}\text{MHz} && (3.4)\\ &= 4\text{GHz}. && (3.5) \end{aligned}$$

What does this number even mean? Consider computer A with 4GHz clock rate and B with 2GHz. The number tells us that computer A can

have 4×10^9 cycles in each second, while B has only 2×10^9 cycles per second. If both A and B are using the CPU we designed in thie chapter, meaning they have five stages, we see that computer A can load twice the amount of instructions than B in each second. Thus, given the same amount of time, A can accomplish more tasks than B.

A more straightforward measurement is simply run the same program on two computers, and see which one runs faster. Let's assume all other factors are the same for both computers, and the only thing that's affecting their performance is the CPU design. The time spent on a program is called **CPU time**. Think about this — to execute a program is basically to execute a sequence of assembly instructions. If a program consists of I instructions, and each instruction needs CPI cycles to finish, and each cycle takes C seconds (the clock period) to finish, the CPU time is simply $I \times CPI \times C$.

Thus, we derive the equation for CPU time of a program:

$$\text{CPU time} = I \times CPI \times C \tag{3.6}$$

$$= \frac{I \times CPI}{\text{Clock rate}} \tag{3.7}$$

where I is the number of instructions in the program, CPI the number of clock cycles per instruction, and C the clock period.

This equation also outlines the factors that can improve a computer's performance:

- For CPI, we prefer smaller number of cycles for each instruction. Take our previous CPU design as an example. For our pipeline design, because each of the five stages costs one cycle, the clock cycle is 5 per instruction. However, in our non-pipelined design, each instruction goes through the entire five stages in one cycle, its clock cycle is 1. So why don't we just use one cycle and non-pipelined design? Because there's another factor in the equation: clock rate;

- For clock rate, the higher the better, because that means given one second we can process more instructions. Again, for our non-pipelined design, because one cycle needs to deal with all five stages, each cycle needs much longer time, which greatly reduces the clock rate. Therefore, for CPU designers, finding a balance between smaller CPI but also larger clock rate is important to achieve better performance. So as programmers, if we cannot improve the hardware design, what can we do? Optimize our algorithms!

- The only factor we can control as programmers is instruction count I. Surely modern compilers are very smart—when they compile our high-level language programs into assembly they optimize a lot to reduce I. However, no matter how smart a compiler is, it can never optimize a linear search to compete with binary search, and that's why algorithms are important to improve performance as well, but from a software perspective.

Example 3.2

We designed a processor A with 2GHz clock rate, and when we run a performance test program on processor A, the CPU time is $10s$. Our competing company designed a processor B, and when we run the same test program on processor B, it takes only $6s$. Our CEO was not happy about this, so he hired a spy to find out what makes their processor so fast. Unfortunately, the company keeps their secret so well, so the only thing the spy can get is that their clock cycles per instruction CPI_B is 1.2 times of our CPI_A. So how fast must computer B's clock be?

Solution: Let CPI_A be the cycles per instruction for processor A. Based on the question, we can easily get the *total* number of instructions in the program:

$$I = \frac{\text{CPU time} \times \text{Clock rate}}{CPI_A} \tag{3.8}$$

$$= \frac{10s \times 2\text{GHz}}{CPI_A} \tag{3.9}$$

$$= \frac{2 \times 10^{10}}{CPI_A}. \tag{3.10}$$

Because $CPI_B = 1.2 \cdot CPI_A$, the clock rate of processor B can be calculated as:

$$\text{Clock rate}_B = \frac{I \times CPI_B}{\text{CPU time}} \tag{3.11}$$

$$= \frac{I \times 1.2 \times CPI_A}{6s} \tag{3.12}$$

$$= \frac{I \times CPI_A \times 1.2}{6s} \tag{3.13}$$

$$= \frac{2 \times 10^{10} \times 1.2}{6s} \tag{3.14}$$

$$= 4\text{GHz}. \tag{3.15}$$

The calculation above is a simplified version, though, because in modern processors, not all instructions take the same number of cycles to finish. For example, a floating point calculation takes longer than an integer one. Therefore, a more detailed measurement should be used.

Based on the different number of cycles an instruction takes, we separate them into different classes. Assume there are K instruction classes, and for a program there are I_k instructions for a $k \in K$. The clock cycles needed for this program can be calculated as:

$$\text{Clock cycles} = \sum_{k=1}^{K} I_k \cdot CPI_k \tag{3.16}$$

The **weighted average CPI** can be calculated as

$$\overline{CPI} = \frac{\text{Clock cycles}}{I} \tag{3.17}$$

$$= \frac{\sum_{k=1}^{K} I_k \cdot CPI_k}{I} \tag{3.18}$$

$$= \sum_{k=1}^{K} \left(CPI_k \cdot \frac{I_k}{I} \right) \tag{3.19}$$

where I is the total number of instructions in the program, and $\frac{I_k}{I}$ is called **relative frequency** of instruction k.

Memory Hierarchy and Virtual Memory

4

In previous chapters we discussed the processor. Now we are going to move on to the memory. Even if it's not part of the processor, it plays a critical role in our datapath and processor design. Thus, in this chapter, we're going to learn about memory technologies.

4.1 Memory Hierarchy

Throughout Chapter 3, we learned that utilizing a pipeline in the datapath can greatly improve the efficiency and throughput of our processor. Specifically, since we separate the entire execution of an instruction into five stages, at its peak, we can have five instructions running at the same time.

Even though pipeline is a great idea, it has its limitations as well. In our previous example, when we introduced the idea of pipeline in Section 3.4.1.1, we *assume* the delay of each stage is equal. This, however, is rarely the case. Typically, some stages take longer time than others. Notice we use one clock signal to control all registers, for the purpose of synchronization of all stages. If some stages take longer than others, the clock has to wait for it, and all the other stages are idle as well.

In other words, the shorter a clock cycle is, the shorter the latency of each instruction. However, to synchronize all stages, the clock cycle is determined by the slowest stage. In Figure 4.1, we assume stage 3 takes the longest time to complete. Therefore, the clock cycle needs to cover the latency of stage 3 in order to synchronize all other stages, and we see that both stages 1 and 2 are idle for quite some time, which is a waste of resources.

This in fact applies to our processor as well. Among the five stages, ME stage where we need to read or write memory is the slowest. It is too slow actually, comparing to reading or writing registers. As shown in Figure 4.2, we see that the CPU cycle time is getting shorter, which is a good thing because that means the delay of each instruction is shorter. However, the

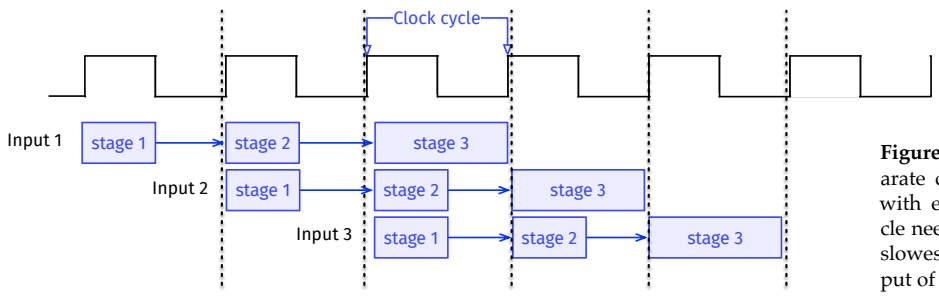


Figure 4.1: It is very challenging to separate combinational circuits into stages with equal latency. Thus, the clock cycle needs to be long enough to cover the slowest stage, which limits the throughput of the entire system.

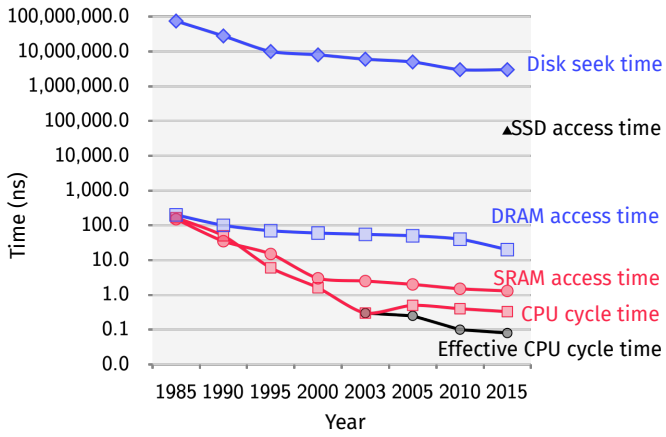


Figure 4.2: The time gap between accessing DRAM/SRAM (the memory) and CPU cycle time is getting larger through the years. Figure borrowed from *Computer Systems: A Programmer's Perspective*.

memory access time (DRAM), didn't get much shorter, and so the gap between CPU and memory is getting larger. [1]

1: The memory we usually talk about is DRAM (dynamic RAM). Later we will also introduce SRAM (static RAM), but it's not used for memory.

To address this, let's start from a programmer's view, and then look at how hardware designers take advantage of this observation to design faster access devices.

4.1.1 Locality of Reference

Let's start with a very simple example, where we assume there's an integer array `a` of length `n`:

```
int sum = 0;
for (int i = 0; i < n; i++) sum += a[i];
return sum;
```

This example sums over all the element in the array `a`, by using a `for`-loop and add each element to the variable `sum` each time. So far we should've been pretty familiar with how arrays are stored in memory. In this example, all the elements in array `a` are stored next to each other. We are also familiar with the calculation happened in the processor, so the variable `sum` is an integer stored in the memory, and each time when we add `a[i]` to it, we need to load `sum` to the processor.

Our memory is a very large device that can store lots of data, but when we look at the example above, we notice:

- The data we're dealing with, *i.e.*, `a[i]`, are stored right next to each other, instead of being all over the place;
- The variables `sum` and `i` are used in each iteration, frequently.

These two characterizes common observations of our programs: spatial locality and temporal locality.

More formally, **spatial locality** means the data with nearby addresses tend to be used (referenced) close together in time, such as the array elements; **temporal locality** means recently referenced data are likely to be referenced again in the near future, such as `sum` in our example.

Language Reference Patterns

Optimizing code with locality in mind is a good practice for programmers. Compare the following two functions in C language. Which one has good locality and why?

```c
int sum_array_rows(int a[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}

int sum_array_cols(int a[M][N]) {
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; j++)
            sum += a[i][j];
    return sum;
}
```

To compare this, we need to know how two-dimensional arrays are stored. C language is a **row-major order language**, meaning for a two-dimensional array, it stores rows next to each other, so the elements in the same row are grouped together. See Figure 4.3 for an illustration. Languages like Fortran are **column-major order language**: they store elements in each *column together* (Figure 4.4).

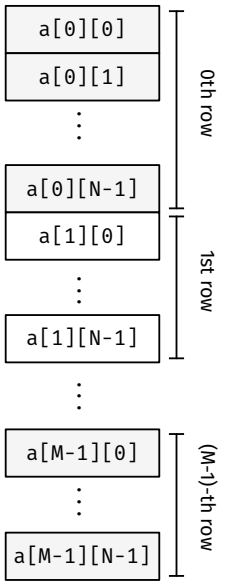


Figure 4.3: Row-major languages store all elements in each row together.

Since we're comparing C functions, we need to look at the row-major arrangement. Apparently, `sum_array_rows()` has good locality, because the inner loop accesses elements stored together. We usually call this **stride-1 reference**.

The inner loop of `sum_array_cols()` needs to skip over N elements in each iteration, so the elements it gets access to are far away from each other. It has a stride-N reference.

However, this is not to say `sum_array_cols()` does not have good locality in every language. In column-major order languages, it has better locality than `sum_array_rows()`. So knowing the language behavior is the first step to determine locality.

Once this observation has been made, smart hardware designers thought, if they are used frequently and together, why don't we just load all of them into another device that's faster than memory just once, and put them back to memory once it's done? That's exactly what caching does.

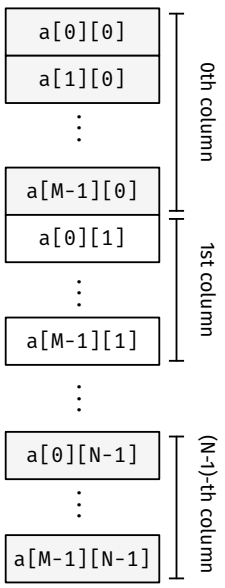


Figure 4.4: column-major languages store all elements in each row together.

4.1.2 Caching and Hierarchy

The idea of caching is not new actually. Think about this: our programs are stored on the hard drive, but why would they be sent to memory when

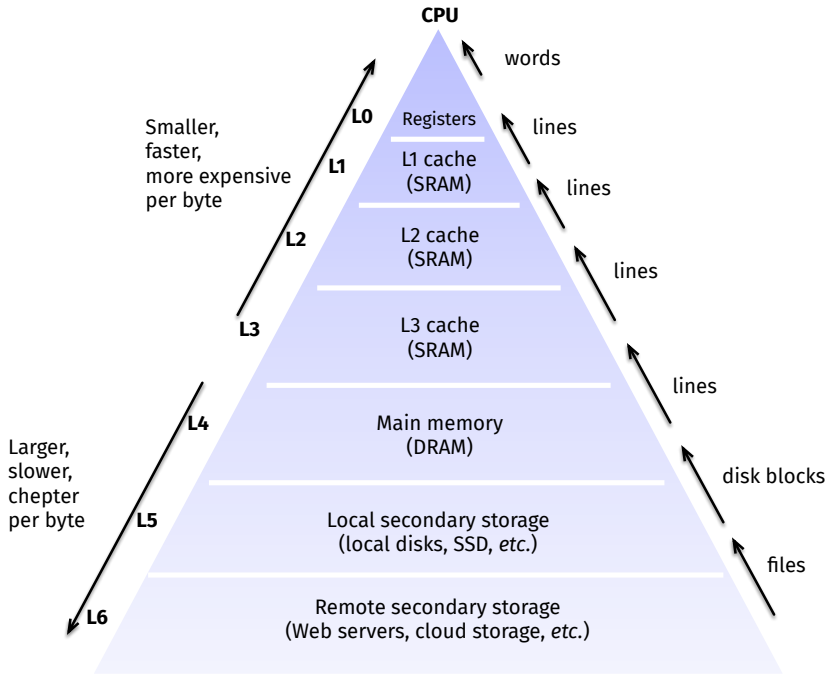


Figure 4.5: Memory hierarchy. Figure borrowed from *Computer Systems: A Programmer's Perspective*.

executing? From Figure 4.1, we see that disk seek time is almost 10^9 times slower than CPU cycle time, so if the data and the code we need to use in a program are stored in the hard drive, the delay would be intolerable. Thus, we store the code and data we're going to use in this program into memory, to accelerate the execution. We can think memory is "caching" the disk.

The reality about storage devices is usually the faster the device is, the more expensive it is to store one byte, and therefore we tend to store less data in that device. Registers are built inside the CPU, and so they are the fastest storage. Due to the cost, however, we cannot have large amount of registers, so in our ARM architecture there are only 32 of them. [2]

If we organize these devices in a picture, we'll have a pyramid-like diagram shown in Figure 4.5, which is usually called **memory hierarchy**. As we see, the device at level k is a cache of the device at level $k+1$. As k increases, the devices are getting cheaper, larger, but also slower. The device at level k stores frequently used data from level $k+1$ as a buffer, so that device at level $k-1$ can get faster access. If the data requested by $k-1$ is not at level k, device k will retrieve it from $k+1$.

Table 4.1 also shows a more detailed hierarchy.

2: Of course the design of CPU is also one of the factors. It's entirely possible to have thousands of registers built in a processor, assuming you're a billionare and can afford them. However, remember if you have those registers, you also need to build an entirely new instruction set architecture to operate them, which is already a large project. Even if you finished it, probably no other people can afford using this processor because it's too expensive. After all, how many billionares do we have in total on this earth?

Table 4.1: A typical memory hierarchy in detail.

Cache type	What's cached?	Where is it cached?	Latency (cycles)	Managed by
Registers	4–8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual memory	4KB pages	Main memory	10^2	Hardware + OS
Buffer cache	Parts of files	Main memory	10^2	OS
Disk cache	Disk sectors	Disk controller	10^5	Disk firmware
Network buffer cache	Parts of files	Local disk	10^7	Web browser
Browser cache	Web pages	Local disk	10^7	Web browser
Web cache	Web pages	Remote server disks	10^9	Web proxy server

4.2 Cache Memory

All of our running programs reside in the main memory, including the variables used in our programs. So far it should be very clear that in order to do any compuation, we need to load them from memory to the registers. However, from Table 4.1 we see that in order to retrieve data from main memory, we have to wait 100 clock cycles. Therefore, we add a device called **cache** on the processor, as a buffer between memory and the processor. [3] This cache is even more expensive than memory to produce, so we can only hold a small amount of data inside it. When we execute a program, the entire program is still stored in memory, but if we're going to deal with an array, we can copy the array into the cache, so CPU doesn't have to go to memory each time when it needs the data.

In this section, we'll start with brief overview of main memory and its operations, and then move on to the cache.

3: The term cache, or caché, has two meanings in our discussion. In general, it simply means "buffer". For example, "memory is a cache of hard drive". It can also mean a specific type of device called cache, which is used specifically between memory and CPU. Since they are based on static RAM (SRAM), we will call them SRAM cache when there could be some confusion. Otherwise we'll simply use the term cache, and it should be clear based on context.

4.2.1 Memory Transactions

The memory we talked about is usually called Random Access Memory (RAM). It's called "random" not because its data are random; it means given an address, we can get the data stored at that address right away. This is different than other type of storage devices, where we'll have to read from the beginning. [4]

4: Think about cassetts, if you know what that is!

RAM is traditionally packaged as a chip, where each chip contains multiple cells. Each cell stores one bit of data. RAM comes in two varieties: static (SRAM) and dynamic (DRAM). The main memory we talk about is formed by connecting multiple such DRAM chips together. Thus, we will also sometimes use DRAM to refer to main memory.

Figure 4.6 shows a typical bus structure between CPU chip and the main memory. The bus interface inside CPU chip allows extension of internal bus (from register file to bus interface) to connect with I/O devices as well as the main memory. There are also some other components in this structre:

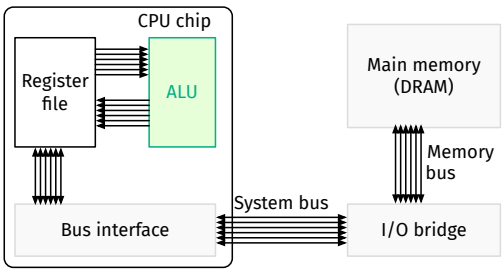


Figure 4.6: Traditional bus structure between CPU chip and main memory.

- **System bus** contains three major parts: control bus, address bus, and data bus;
- **I/O controller** connects I/O devices to the system bus to be controlled and used by CPU chip;
- **Memory bus** contains address, data, and control bus as well, which has been covered earlier in Chapter 3. The role of control bus here is to indicate if this is a read or write transaction.

Read Transaction

The read transaction between CPU and the main memory is mostly done by instructions such as `LDUR`. For example, given an instruction `LDUR X10,[X9]`, the value of `X9` is read from the register file first. Then it's put on the system bus and memory bus through bus interface and I/O bridge. Next, main memory retrieves the data stored at the address, and puts the data on memory bus, then transfers back through system bus and writes to register `X10`.

Write Transaction

Similarly, `STUR` instruction invokes write transaction. For example, `STUR X10,[X9]`, the data in `X9` will be put on address bus, and that in `X10` on data bus transferred to the main memory.

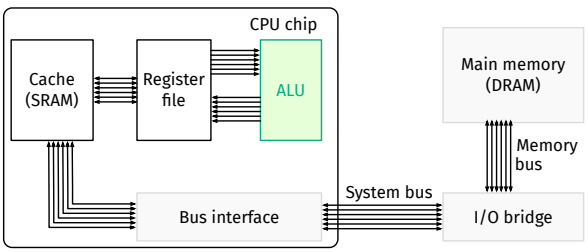
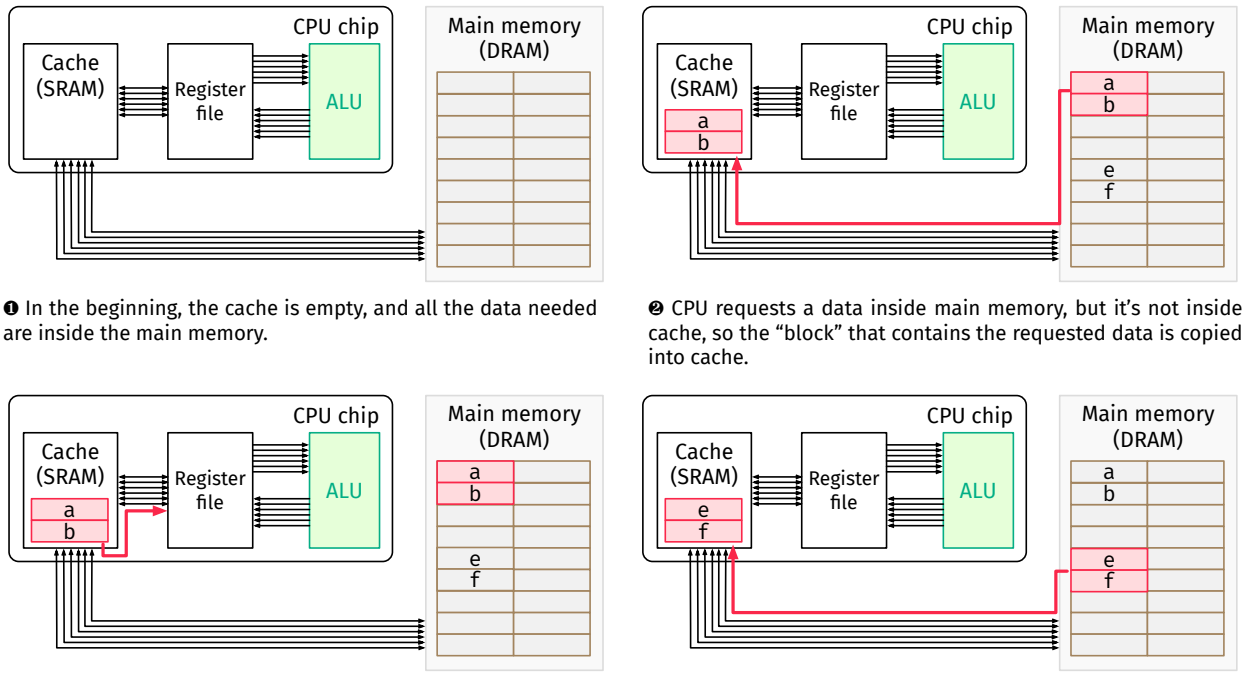


Figure 4.7: A CPU chip with one level of cache.

4.2.2 Adding Cache to the Processor

Since memory transactions are too slow for the processor, we add another type of memory called **cache** on the CPU chip. This cache is formed by a collection of Static RAM (SRAM), which is more expensive but faster. It's also smaller than the main memory, so it can only hold a tiny subset of the

5: Through the years, engineers realized that one level of cache is not enough, so they created three levels of cache, named L1, L2, and L3 cache, as in Figure 4.5. In fact there are more levels, but here one is enough for understanding the concept.



❶ In the beginning, the cache is empty, and all the data needed are inside the main memory.

❷ CPU requests a data inside main memory, but it's not inside cache, so the "block" that contains the requested data is copied into cache.

❸ As long as the data requested by CPU are in the cache (called hit), we don't need to go to main memory anymore, and can simply go to cache to retrieve the data.

❹ If CPU requests a data that's not in the cache (called miss), the "block" that contains the data in the main memory will be copied into cache, and overwrite the cache.

Figure 4.8: An illustration of how cache works in a toy example.

data we use in a program. [5] In Figure 4.7, we added a cache on the CPU chip between register files and the bus interface.

Let's start with a toy example to see what this cache actually does. As in Figure 4.8, in the beginning, cache is empty, and all the data needed are inside the main memory. Assume now the CPU executes an instruction `LDUR X0,[X1]`. Because the data `M[X1]` is not in the cache, we have to go to the main memory to retrieve it. The thing here is, when retrieving data from memory, we don't just bring the requested data back; instead, we bring a "block" of data back. In other words, in addition to copy the requested data back, we also copy the data stored next to it back to the cache. For example, if we want to load the double word at address `0x1000` to `0x1008`, we copy all the data from `0x1000` to `0x1040` back to the cache, which contains eight double words.

Next time, as long as the data requested by CPU are in that address range (`0x1000`—`0x1040`), since they are already copied in the cache, there's no need to go to the main memory anymore; we can simply retrieve the data from the cache. When this happens, we say it's a **hit**.

However, if the data requested is not in the cache, we'd have to go to main memory again, and copy the "block" back to the cache, and possibly overwrite the data in the cache. The case where the data requested is not in the cache is called a **miss**.

Now with this idea, it is not wonder why we prefer good spatial locality in our programs. Cache stores a chunk of memory data each time, so if every

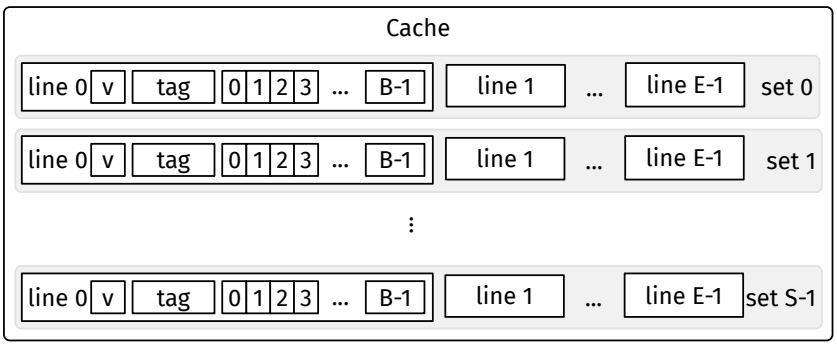


Figure 4.9: A cache with S sets, E lines per set, and stores B bytes for line. Note all the lines in the cache have the same structure as shown in line 0; due to illustration purposes we only show structures of line 0.

`LDUR` in our program is requesting data that are close to each other, we can easily and rapidly get them just from the cache, without going further to the main memory. This is also why we prefer stride-1 reference in arrays, because cache copies consecutive elements in an array all at once.

4.2.3 Cache Organization

The "blocks" we mentioned earlier, technically, is called a **line** in cache. In Figure 4.9 we show a general organization of a cache. A cache can be characterized by three parameters: S, E, and B. Each cache has $S = 2^s$ **sets**, where each set has $E = 2^e$ lines. In each line, there's a **valid bit** `v` to indicate if the data stored in this line is valid or not. [6] A **tag** is also used for identification of the data. [7] Then there are $B = 2^b$ bytes that store the actual cached data we copied from the main memory. Thus, we see that the capacity of the cache is $S \times E \times B$ bytes.

6: For now we can just assume if there's data stored in the line, it is valid, so the valid bit is set. Otherwise, it's invalid, and the valid bit is clear.

7: If this is not clear for now, keep reading, and we'll have an example soon to help you understand.

As we know, when CPU executes an instruction such as `LDUR X1,[X0]`, it's requesting the data stored at the address indicated by `X0`. When we have a cache between the processor and the main memory, the first step in this case is to see if the data is already in the cache. So given an address issued by CPU, how can we get the corresponding data from the cache?

In fact, the address sent out by CPU can be chopped into three fields:

t bits	s bits	b bits
tag	set index	block offset

The first t bits are used for the tag; next s bits for indexing to a specific set; and last b bits for block offset.

Given an address, there are several steps to retrieve the data:

- Use set index in the address to locate to a specific set;
- Compare the tags in all the lines in that set with the one in the address:
  - If there's a line whose tag matches ours, we have a **hit**. The data is then located in that line, starting from block offset;
  - If none of the line has matching tags, we have a **miss**. Then we'll need to go to the main memory, and copy the data back.

Quick Check 4.1

Assume we have three caches with different organization parameters. In the following table, m is the number of bits of a memory address, while C the capacity of the cache (in bytes). The rest of the parameters are the same as we used in this section. Please fill in the table below.

Cache	m	C	B	E	S	t	s	b
1	32	1,024	4	1				
2	32	1,024	8	4				
3	32	1,024	32	32				

In the following sections, we will discuss two special types of caches, and use concrete examples to see how cache works.

4.2.3.1 Direct-Mapped Cache

The simplest cache is called **direct-mapped cache**, where each set has only one line, *i.e.*, $E = 1$. We'll create a toy direct mapped cache to show how it works. In this toy example, we assume:

- The main memory uses 4-bit addresses, so it can store 16 bytes of data. Each byte has an unique address;
- Our mini cache is direct mapped, so one line per set;
- The cache has in total 4 sets, so $s = 2$;
- Each set can store 2 byte of data, so $b = 1$;
- Each time we retrieve one byte of data.

Because each address has 4 bits, we use $s = 2$ bits for set index and $b = 1$ bit for block offset, which leaves us one bit for the tag, *i.e.*, $t = 1$. We'll also follow the convention from last chapter, and use `M[x]` to denote the data in memory at address of `x`.

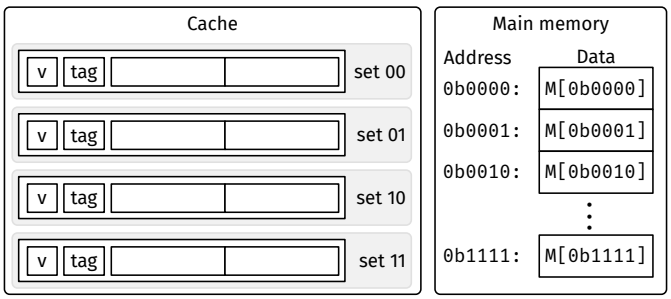
Now let's start requesting data! Assume the CPU requests data at the following addresses one at a time: [8] `0b0000`, `0b0001`, `0b0111`, `0b1000`, `0b0000`.

8: You can think that we are running a sequence of assembly instructions such as `LDURB W1,[0b0000]`, `LDURB W2,[0b0001]`, *etc.*

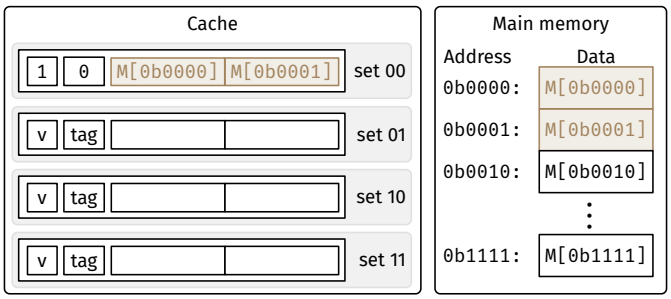
We first parse their addresses into tag, set index, and block offset as follows:

Address	Tag	Set index	Block offset
`0b0000`	`0`	`00`	`0`
`0b0001`	`0`	`00`	`1`
`0b0111`	`0`	`11`	`1`
`0b1000`	`1`	`00`	`0`
`0b0000`	`0`	`00`	`0`

In the beginning, the cache is empty:



`0b0000` The first address requested is `0b0000`. Based on our discussion, we have a tag `0b0`, set index `0b00`, and block offset `0b0`. Set `00` has nothing in it, so we have a miss. We'll go to memory address `0b0000` to retrieve both `M[0b0000]` and `M[0b0001]` back, because each line in the cache and store two bytes. After this, our cache looks like this:

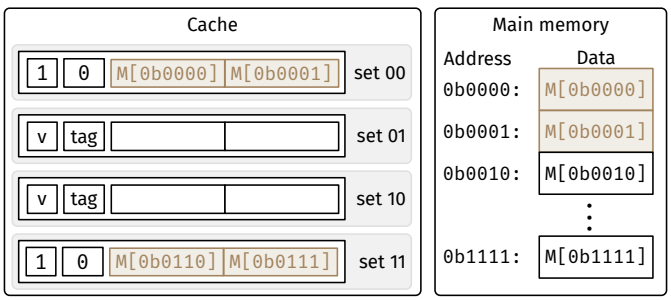


Now that we have the data in the cache, we will load one byte starting from offset 0 in the block. Thus, we take `M[0b0000]` back to register files.

`0b0001` The second address is `0b0001`. Its set index is `0b00`, so we index into set `00`. At this point, we notice the valid bit is set, and its tag `0` matches the tag from our address, so we have a hit! Next, the block offset is 1, so we'll retrieve the data from the second byte in the line, which is `M[0b0001]`.

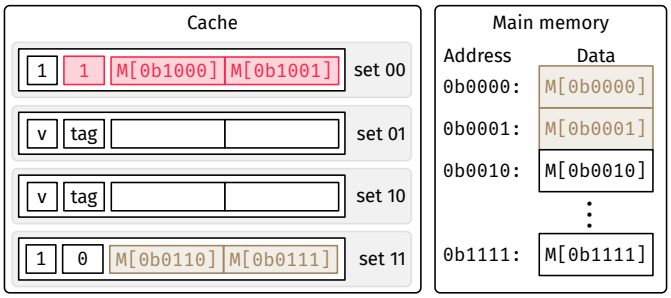
`0b0111` This time the set index becomes `0b11`, so we have a miss. We go to the main memory, and copy both `M[0b0110]` and `M[0b0111]` back to the cache. [9] After copying the data from the main memory, we set valid bit, and set tag to 0. Since the block offset is 1, we again load the second byte in the line, `M[0b0111]`, to the register file. The cache at this point looks like this:

9: Note here we didn't copy `M[0b0111]` and `M[0b1000]` back, because remember the last bit in an address in this example indicates the block offset. Given an address, we want to copy the data at addresses with the **same** tag, **same** set index, but from block offset of 0. This way we can carry an entire line into the cache.



`0b1000` This address has a set index of `0b00`, so we index into set `00`, and

proceed to compare the tag. The tag in the set `00` is 0, which doesn't match the tag in our address, so the data currently in this set is not the one we want, and we have a miss. We need to go to the main memory, and copy `M[0b1000]` and `M[0b1001]` back. Since each set has only one line, we'll have to **evict** the data currently residing in the set, and replace it with the new data we just brough back from the main memory. Thus, `M[0b0000]` and `M[0b0001]` are overwritten by `M[0b1000]` and `M[0b1001]`, and the tag is updated to 1. After these operations, the cache looks like this:



Then we can send the first byte from the line back to the processor.

`0b0000` Lastly, this address again indexed to set `00`, but from last address we replaced the data in set `00` with tag 1, so we have a miss again, unfortunately. We'll have to repeat the procedure described above: bring the data back from the main memory, overwrite the current data in the set, and update the tag.

4.2.3.2 A Real-World Example

In the following, to help you understand the concept, we'll use a real world example. Of course, if the procedure of cache read we discussed in the previous section is clear to you, you can skip this section.

Let's say you're working at a casino, and your job is to provide one of the eight cards to a customer: ♥`00`, ♥`01`, ♥`10`, ♥`11`, ♦`00`, ♦`01`, ♦`10`, and ♦`11`.

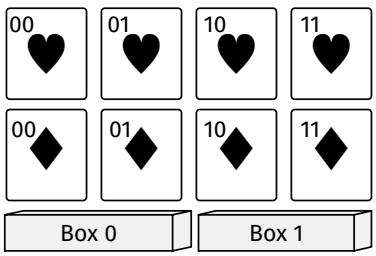


Figure 4.10: For convenience, two boxes are used for storing two cards each. Box x only stores cards starting with digit x.

Those cards are stored in the stockroom but you're working at the front desk. You realized that it'll be too much hassle if you go back to the backroom every time a customer comes and asks for a card. You're very smart, so you prepared two boxes at the front desk: box 0 and box 1. When a customer asks for a card, you check if it's already in the box or not. If it is, then you just give it to the customer; otherwise you go back to the stockroom, bring it back, and put it in the box.

The rule of using the box is simple. The first digit on the card represents which box. For example, if a card is `0x`, you put that into box 0; if it's `1x` you put it into box 1. Because the box can fit two cards at a time, you decided to bring two cards with the same first digit back to the box. Thus, the second digit represents which card. For example, `x0` means it's the first card in the box; `x1` the second card.

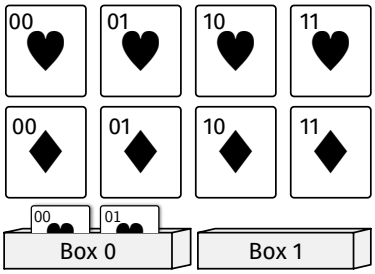


Figure 4.11: After ♥00 was requested.

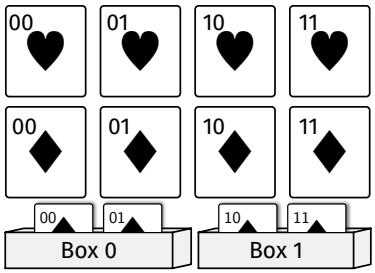


Figure 4.12: After ♦01 was requested.

Now let's get started!

♥00 The first customer asked for a ♥00 card. Since you just started working, there's nothing in the boxes—you had a *miss*. You went back to the stockroom, and brought both ♥00 and ♥01 back and put them in box 0, because both of them start with 0. ♥00 means it's the 1st card in box 0, you grabbed it and handed it to the customer. The customer used the card and gave it back to you;

♦11 The next customer asked for a ♦11 card. You checked box 1 but nothing there—you had a miss again. You went back to the stockroom, and grabbed both ♦10 and ♦11 back, because both of them start with 1. After putting them in the box, you realize ♦11 is the second card in box 1, so you took it to the customer;

♥01 This customer wanted a ♥01. First thing you checked is box 0, and yes there are cards there. Then you need to make sure the cards there are in the ♥ suit, and fortunately they are, so we have a hit! You gladly picked the second card in box 0, and handed it to the customer. No need to go back to the stockroom!

♦01 This time, a customer wanted a ♦01. You went to Box 0, and there were some cards there indeed. However, when you compared their suits, you realized the one in the box (♥) is not what's requested (♦), so you had a miss again. You had to go back to the stockroom, and grabbed both ♦00 and ♦01 back. The box 0 can only fit two cards, so you swapped the two cards in the box out, with the new cards you brought.

In this example, a customer's request contain three information: suit, box number, and card number. If we treat the requests as three-bit addresses, and the card as the data requested, we have the perfect analogy:

- suit → tag;
- box number → set index;
- card number → block offset.

4.2.3.3 Associative Cache

In direct-mapped cache, we see that when there's a conflict (the tag of the address doesn't match the one in the line), we have to go to the main memory and replace the data in the cache. You might want to ask: what if for each set, instead of having only one line, there are two lines? One line has a tag of 0, while the other has 1. In that case, we don't need to evict any line, and both `M[0b0000,0b0001]` and `M[0b1000,0b1001]` will be placed in the cache. This is actually the exact idea behind associative cache.

Table 4.2: Notations in cache.

C	Cache capacity in bytes
S	Number of sets
E	Number of lines per set
B	Number of bytes per line

As discussed earlier, a cache can be characterized by a tuple of parameters: (S, E, B), and the capacity in bytes $C = S \times E \times B$. See Table 4.2 for a small summarization. Given a fixed capacity C, we can adjust the three parameters to have different types of caches. When $E = 1$, meaning each set has only one line, we have direct-mapped cache; when $S = 1$, we have a **fully associative cache** where we only have one line; any other combinations of S, E, and B is called **E-way set associative cache**.

Let's look at an example. Assume the memory addresses are of 4-bit, and the total capacity of the cache $C = 8$. We also assume each line can store two bytes of data, and so the LSB of the memory addresses is always block offset. The three possible cache organizations are shown in Figure 4.13.

- **Direct-mapped cache:** Because $E = 1$, we'll have $C = 8 = S \times B$. Because $B = 2$, we have $S = 4$ sets in total. Thus, for a four-bit address, MSB is the tag, and LSB the block offset, and the middle two bits are set indices;
- **2-way set associative cache:** We can reduce the number of set by half, which makes $S = 2$ sets. To keep $B = 2$ bytes per line, we have to make two lines per set to keep the capacity unchanged. Thus, $S = E = B = 2$. Now that we only have two sets, we'd need one bit for set index. LSB is still the block offset, so the most significant *two* bits are used for the tag;
- **Fully associative cache:** We have only one set in this case, so there's no need to use any bit in the address as set index. To keep capacity unchanged, we need four lines, and since LSB is the block offset and we don't need set index, the most significant three bits of the addresses are used for the tags.

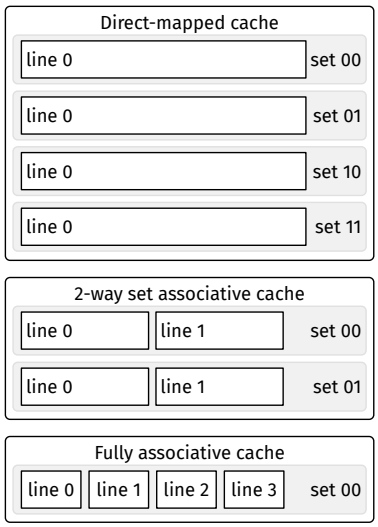


Figure 4.13: Three possible cache organizations with a total capacity of eight bytes, and with lines that can store two bytes of data each.

There are two possible problems coming with E-way set associative caches. Take the 2-way cache above as an example.

The first problem is if the data requested is not in the cache and there are empty lines in the set, which line should we put the data in? A simple solution is just to put it in the next available line.

Another problem is when there's a conflict. Because each line has two bits for the tag, there are four possible tags for each set in total. However, because we only have two lines per set, there's a chance that both lines' tags don't match the tag in the address. So we need to replace one line, but which line?

There are several simple algorithms. For example, we can take the earliest used line out, assuming it won't be used again very soon. We can also take a random line out and hope for the best. In our class, we'll replace the earliest used line.

Quick Check 4.2

The following problem concerns basic cache lookups.

- The memory is byte addressable;
- Physical addresses are 13 bits wide;
- The cache is 2-way set associative, with a 4 byte line size and 16 total lines.

In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

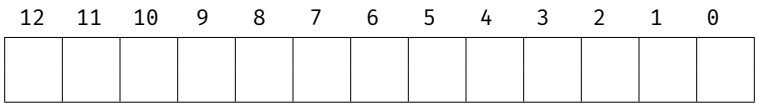
2-way Set Associative Cache

			0	1	2	3			0	1	2	3
Set	Tag	V	Bytes				Tag	V	Bytes			
0	09	1	86	30	3F	10	00	0	99	04	03	48
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	2F	81	FD	09	0B	0	8F	E2	05	BD
3	06	0	3D	94	9B	F7	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	B0	39	D3	F7
6	91	1	A0	B7	26	2D	F0	0	0C	71	40	10
7	46	0	B1	0A	32	0F	DE	1	12	C0	88	37

Part 1

The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

O The block offset within the cache line
I The cache index
T The cache tag

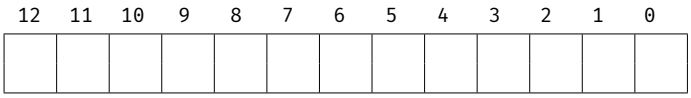


Part 2

For the given physical address, indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs. If there is a cache miss, enter "-" for "Cache Byte returned".

Physical address: 0x0E34

1. Physical address in binary (one bit per box)



2. Physical memory reference:

Parameter	Value
Byte offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

4.2.3.4 Write Transactions

The operations we discussed above are limited to read transactions, *i.e.*, LDUR instructions, since they are easiest to demonstrate the idea behind

caches. In this section we will briefly discuss write transactions, but it's not our focus.

The main issue with write transactions is due to multiple copies throughout the entire system. Think about this: suppose the processor executes an instruction `STUR X0,[0x1000]`, which needs to write data stored in `X0` to memory address `0x1000`. If the data stored at this address has already been copied into the cache, now the problem is: do we only update the cache or only the memory, or both?

If we only update the cache, we will have two different values of the same address in the cache and the memory. If later this cache line was replaced, the new value will be gone, and the memory still stores the old value.

If we only update the memory, the delay is too long, but also there'll be data inconsistency. If the next instruction is `LDUR X0,[0x1000]`, because the data is already in the cache, we will only load cache into the processor, and it'll be the old value.

For write transactions we also have hit and miss, *i.e.,* if the destination of `STUR` instructions has already been copied into the cache or not, and the rule is the same as read transactions.

We have two ways to deal with write hit:

- **Write-through**: writes directly to the main memory as well as the cache;
- **Write-back**: updates the cache only first, and only updates the main memory when the line is replaced.

We also have two ways to deal with write miss:

- **Write-allocate**: copy the line into the cache from the main memory first, and then update the data in the cache;
- **No-write-allocate**: writes straight to the main memory without loading into the cache first.

Typically, designers follow a specific combination when dealing with write transactions. The common combinations are write-through + no-write-allocate, and write-back + write-allocate. For the first option both hit and miss cases deal with the main memory, while the second option deals with cache first only, and only interacts with the main memory when the cache line is replaced.

4.2.4 Multi-Level Caches

As mentioned before, one level of cache between the main memory and the processor is good enough for our understanding of the concept of cache, but it's not practical in reality. Therefore, we see that in Figure 4.5 that we have three levels of L1, L2, and L3 caches. Lk cache holds parts of data copied from L$k+1$ cache as a buffer, and Lk is smaller, faster, and more expensive. Let's take a quick look at how ARM chips organize multi-level caches in Figure 4.14.

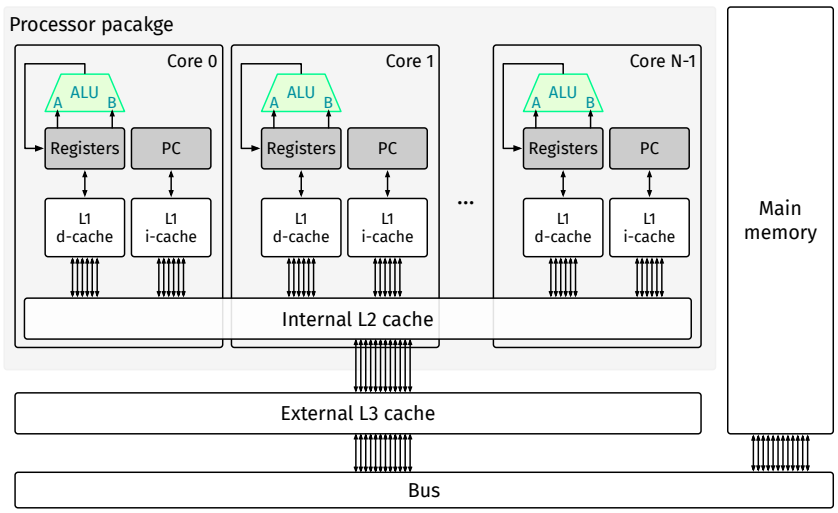


Figure 4.14: Simplified illustration of ARM processor chip with multi-level caches.

You usually see "8-core" or "quadcore" when you buy a computer. Each core is just a CPU that contains the basic elements we have learned. In modern architectures, L1 cache is actually splitted into two, called d-cache for caching data and i-cache for instructions. This further accelerates data processing than just using one L1 cache for both data and instructions.

When we have multiple cores on one processor package, we also have a L2 cache that connects all the cores. It's larger and a bit slower. Sometimes you see some computers contain multiple processor packages, so they can be further connected by one L3 cache. We call this "external" because it's not on any processor chip.

4.2.5 Evaluation Metrics

To evaluate cache performance, the most commonly used one is **miss rate**:

$$\text{Miss rate} = \frac{\text{Times of data not found in the cache}}{\text{Total memory reference}} \tag{4.1}$$

The best way to understand miss rate and how to calculate it is through a concrete example.

Example 4.1

A bitmap image is composed of pixels. Each pixel in the image is represented as four values: three for the primary colors (red, green and blue – RGB) and one for the transparency information defined as an alpha channel.

Assume we will use a direct-mapped cache of 128 bytes with 8-byte blocks. The definition of a pixel and the matrix we're going to use is defined as follows:

```c
typedef struct{
    unsigned char r;
    unsigned char g;
    unsigned char b;
    unsigned char a;
} pixel_t;

pixel_t  pixel[16][16];
```

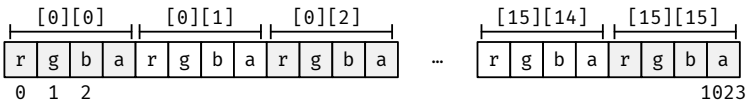
Also assume that

- `sizeof(unsigned char) == 1`;
- `pixel` begins at memory address `0`;
- The cache is initially empty;
- Variables `i`,`j` are stored in registers and any access to these variables does not cause a cache miss.

What's the miss rate for writes to the pixel given the following code?

```c
for (i = 0; i < 16; i ++){
    for (j = 0; j < 16; j ++){
        pixel[i][j].r = 0;
        pixel[i][j].g = 0;
        pixel[i][j].b = 0;
        pixel[i][j].a = 0;
    }
}
```

Before starting calculating the miss rate, we need to determine the memory and cache structure. Notice that the code shown is in C, which is a row-major order language, so we can depict the storage of matrix `pixel_t` as follows:



Since it's assumed that the matrix begins at memory address `0`, we see that the last element of the matrix resides at memory address $16 \times 16 \times 4 - 1 = 1023$ (decimal).

One another important thing we need to determine is the cache organization. The cache we're going to use is direct-mapped, so $E = 1$. Each line has $8 = 2^3$ bytes, so $B = 3$. With the total capacity $C = 128 = 2^7$ bytes, we have $S = 4$ sets.

Now let's see how to calculate the miss rate for this example. In the inner loop, we have four writes (assigning zeros to the members of the struct), so each inner iteration we have four memory access in total. Also notice that because each element takes four bytes,

while each line in the cache can store eight bytes, we can store two consecutive elements in each line. We denote them as `[i][j]` and `[i][j+1]`.

In the loop of `j`, the first write, `pixel[i][j].r = 0`, will definitely have a miss, either due to empty cache in the beginning, or line replacement. After this miss, all eight bytes starting from `pixel+i+j` will be carried into the cache, meaning all the following seven writes—`g,b,a` for `[i][j]` and `r,g,b,a` for `[i][j+1]`—will have hit. So we can certainly re-write the code to the following and it's identical in terms of cache and memory access:

```
for (i = 0; i < 16; i ++){
    for (j = 0; j < 16; j += 2){
       pixel[i][j].r   = 0; // miss
       pixel[i][j+1].r = 0; // hit
      pixel[i][j].g = 0; pixel[i][j+1].g = 0; // hit
      pixel[i][j].b = 0; pixel[i][j+1].b = 0; // hit
      pixel[i][j].a = 0; pixel[i][j+1].a = 0; // hit
    }
}
```

What we can conclude from the code above is, for every eight memory access in the inner loop, we will have one miss. The outer loop doesn't really matter here, because it's simply scaling the access by 16. Therefore, clearly the miss rate is $\frac{1}{8} = 0.125$ or 12.5%.

Quick Check 4.3

Given the follow chunk of code, analyze the miss rate given that we have a byte-addressed computer with a total memory of 1 MB. It also features a 16 KB direct-mapped cache with 1 KB blocks. Assume that the cache begins cold (empty).

```
#define NUM_INTS 8192    // 2^13
int A[NUM_INTS];         // A lives at 0x10000
int i, total = 0;
for (i = 0; i < NUM_INTS; i += 128) {
    A[i] = i;            // Line 1
}
for (i = 0; i < NUM_INTS; i += 128) {
    total += A[i];       // Line 2
}
```

1. How many bits make up a memory address on this computer?
2. How many bits are used for Tag, Index, and Offset in the cache?
3. Calculate the cache miss rate for the line marked Line 1;
4. Calculate the cache miss rate for the line marked Line 2.

4.2.6 Writing Cache-Friendly Code

You might be wondering: "sure, theoretically we know how cache works and so on, but does that really make a difference? For all the programs we've written it seems that everything just happens so fast." On the other hand, probably the only difference you've noticed is the Big-O notation learned in algorithm class, where you were taught that the time complexity increases as you have larger inputs.

When you were learning Big-O notation, recall in the beginning a big assumption is that hardware differences are ignored, and so it's a purely theoretical abstraction. Remember, however, that the programs are not running in your imagination; it eventually relies on the hardware on your laptop! Since we have learned calculating miss rate, why don't we see an example, where the programs have identical time complexity but different miss rates, and see in reality what their performance is.

Our task is very simple—matrix multiplication, given two `long int` matrices `a` and `b` of same size $n \times n$. Each element takes eight bytes. Based on the math definition, we can quickly write out a segment that performs the multiplication:

```
/* ijk */
for (i = 0; i < n; i ++)  {
    for (j = 0; j < n; j ++) {
        sum = 0.0;
        for (k = 0; k < n; k ++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Another way to perform matrix multiplication is to fix each element `a[i][k]` in matrix `a`, and iterate over each row in matrices `b` and `c`:

```
/* kij */
for (k = 0; k < n; k ++) {
    for (i = 0; i < n; i ++) {
        r = a[i][k];
        for (j = 0; j < n; j ++)
            c[i][j] += r * b[k][j];
    }
}
```

Or, we can fix each element in matrix `b` while iterate over each column in matrices `a` and `c`:

```
/* jki */
for (j = 0; j < n; j ++) {
    for (k = 0; k < n; k ++) {
    r = b[k][j];
    for (i = 0; i < n; i ++)
```

```
            c[i][j] += a[i][k] * r;
        }
    }
```

The three methods above are mathematically equivalent, and have the same complexity of $O(n^3)$. Let's analyze their miss rates first. Assume the block size of the cache is 32 bytes and is not large enough to store multiple rows. We also assume the dimension n of the matrices is very large, meaning each row needs to be moved into the cache multiple times.

`ijk` The inner loop accesses elements in `a` and `b` each time. Notice for `a[i][k]`, the row-number is fixed while column number `k` constantly changes from `0` to `n-1`. Also because the block size of the cache is 32 bytes and each element takes 8 bytes, we can hold four elements each time. For every four elements, the first one will be a miss due to empty cache or line replacement, while the rest of the three will be hits. Therefore, the miss rate for matrix `a` is $1/4 = 0.25$. As to matrix `b`, notice it gets access to each element *column-wise*, so the miss rate is simply 1. The inner loop does not involve matrix `c`, so we can ignore it.

`kij` In this code, we first notice that the inner loop does not involve matrix `a`, so we skip it. Element `b[k][j]` is iterating over all the elements in each row, so its miss rate is simply $1/4 = 0.25$ (see analysis in the `ijk` case for matrix `a`). Similarly, matrix `c` iterates over all elements in a row, so its miss rate is also 0.25.

`jki` Lastly, for `jki` case, matrix `b` is not present in the inner loop, so we ignore it. For both matrices `a` and `c`, the inner loop iterates over all the elements in one column. You've probably already sensed that this is a bad idea, and yes the miss rate for both is 1.

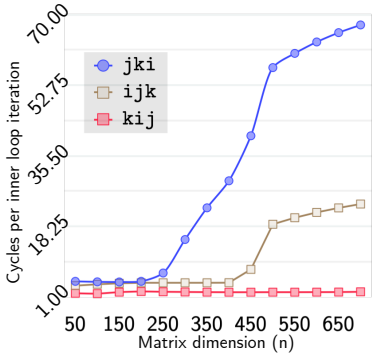


Figure 4.15: Matrix multiplication on Core i7. Large missing rate leads to more cycles needed to run each inner loop iteration.

If we take average miss rate of the three methods, we will have 0.625 for `ijk`, 0.125 for `kij`, and 1 for `jki`. So in terms of miss rate, it's obvious that `kij` is the best way and `jki` the worst. But is that really the case?

The authors of *Computer System: A Programmer's Perspective* have run the code above on an Intel Core i7 machine, and recorded the number of cycles it takes for each inner loop iteration, as the matrix dimension n goes up. Figure 4.15 shows the result. As expected, the larger miss rate, the more cycles it needs per each inner loop iteration. As n increases, the cycles needed also increases. Notice how `kij` almost stays the same.

Of course an efficient algorithm of low complexity is still desired, but remember it's still an abstraction; eventually the computation happens in the hardware. An experienced programmer should not only write code with low complexity, but have the knowledge about the cache and write cache-friendly code.

4.3 Virtual Memory

In previous section, we see that the CPU requests a data by issuing a memory address, and the address is parsed to check the cache first. The addresses received by the cache are actually **physical address**—meaning those are real address used in the real main memory.

Why do we emphasize *physical*? Our laptops are usually now 16GB or at least 8GB of RAM, which is the physical size of the main memory, shared by *all* programs running in the system. Even if 16GB seems a lot, it is far from being sufficient to run multiple real applications on our laptops. For example, right now, just my music player on macOS itself needs 392GB of memory space, not to mention all other apps running on my laptop that are using the same main memory. My memory is only 16GB, so how is a music player that needs 392GB of memory even able to execute? And how do all the concurrently running programs squeeze into a memory as small as 16GB?

Again, let's think about **locality**. We mentioned that locality is an important feature of computer programs, which doesn't only apply to the program data, but also program code itself. For example, given a small amount of time, the assembly instructions running in the program tend to cluster together (they are stored close to each other). Even if we have branching instruction, you don't always jump all over the place, right? Notice two key points there: "a small amount of time", and "cluster". Let's start with a motivating example.

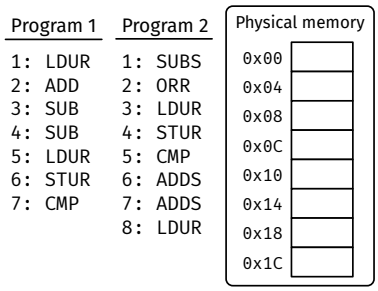


Figure 4.16: Two programs need 60 bytes of memory space, but the actual memory has only 48 bytes.

4.3.1 A Motivating Example

Assume our memory can store only 48 bytes only, as in Figure 4.16, but we want to run two programs at the same time. For the two programs shown in Figure 4.16, we see that there are in total 15 instructions. If each instruction takes 4 bytes, to load them all into the physical memory we need $15 \times 4 = 60$ bytes at least. So how to fit both programs into a small physical memory?

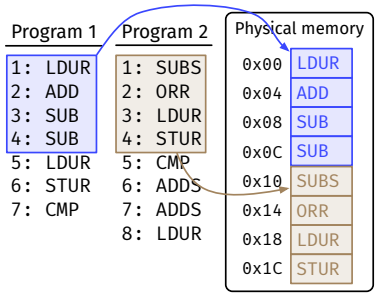


Figure 4.17: Only loading *parts* of the programs allows them to fit in to small physical memory at the same time.

Because of locality, we know that the executed instructions of a program in a small amount of time are usually stored next to each other in memory. So why don't we just load *parts* of the program into the memory first? During that small amount of time, just that part of the program is enough to execute. As shown in Figure 4.17, we only load program 1's and program 2's first four instructions into the physical memory, which will fill the entire memory nicely. When the loaded instructions have all been executed, we take them out, and swap in the other parts of the programs, so we'll end up in a situation as in Figure 4.18.

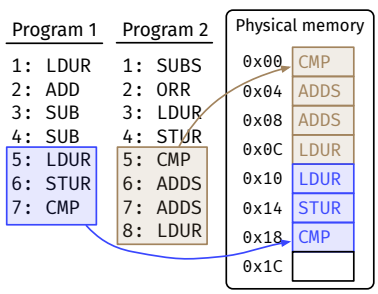


Figure 4.18: When other parts of the programs are needed, we just swap them in.

This is such a simple thing we do almost every day. There might be lots of books at your home, but your bag can only take probably five books max, so before you go to campus, you'd take only the books you need that day to bring. If next day you're going to use a different book, you'll take out

the one you're not going to use, and put the new book into your bag. The books are like parts of the program, and the bag is the physical memory.

4.3.2 Definitions

The example in the previous section is a very simple one, but it illustrates the idea behind virtual memory well. Remember, however, the example is only simplified. In fact, the areas of a process—text, data, stack, and heap—will all be separated into "parts" and loaded into the physical memory, not just the text segment.

We, as programmers, write our code without even thinking about memory space limit. This is because the operating system provides a memory management mechanism called **virtual memory**. From our point of view, each of our program can take basically infinitely large memory space. [10] So all the statements involving "memory address" so far (except the one in the cache), *e.g.*, "pointers represent memory addresses", "`LDUR` loads data at a memory address to a register", *etc*, refer to virtual memory. If a virtual memory address has n bits, **each program** will have the same set of $N = 2^n - 1$ unique virtual addresses $\{0, 1, ..., 2^n - 1\}$, which is called a **virtual address space**.

10: Technically not *infinitely*-infinitely large; the typical virtual memory size for a program is 4GB on Linux, but can certainly be extended as needed.

From the machine's view, however, the physical memory is small and limited. The physical memory is also byte-addressed, meaning each byte has an address. We call this address **physical address**—a real address, not virtual, not imaginary. If a physical address has m bits, the set of $M = 2^m - 1$ unique addresses $\{0, 1, ..., 2^m - 1\}$ is called **physical address space**. Based on our analysis, we notice that $N >> M$.

Obviously, to fit all the programs into one small physical memory needs some arrangement, as we showed in Section 4.3.1. Briefly, we chop each program's virtual memory space into smaller pieces, called **virtual pages**, and only load some virtual pages into the physical memory. When they reside in the physical memory, they are called **physical pages**.

Let's look at Figure 4.19 for an illustration. In this example, we have two processes share one physical memory. Each of the processes has a complete virtual memory space, from 0 to $N - 1$. Their virtual memory spaces also have complete organizations including text and data segment, stack, and heap. This is what *we* see for each process.

Due to space limit, it's impossible to fit all of these two processes into one physical memory, so we chop the virtual memory spaces into pages, and only the data and code in a page is needed do we move the page into the physical memory. In the figure, we see that two pages of process 1 and three pages of process 2 are currently being used. If the processor needs data or code in a page that's not currently in the physical memory, it needs to find a page to swap out. If it couldn't find such a page, then delay will happen. Remember when you open too many apps on your laptop and it got stuck? Yeah that's basically what happened.

Notice some details in the figure:

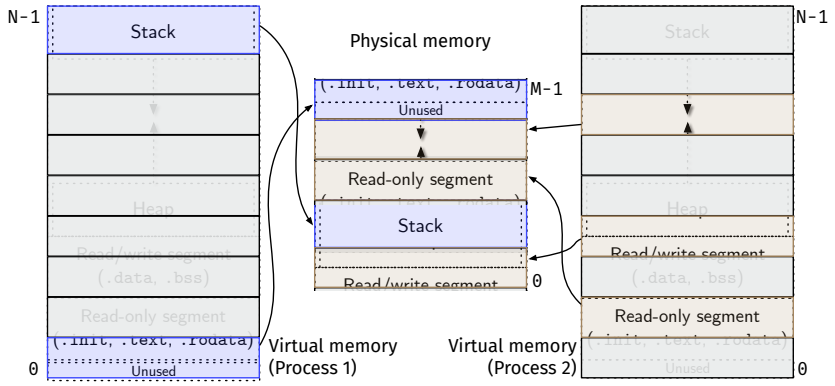


Figure 4.19: To fit multiple processes' virtual memory space into a small physical memory, the system needs to split virtual memory spaces into pages, and only load the pages that contain needed data and code into physical memory.

- Physical pages are not dedicated to a specific process, and that's why we see the pages from two processes are all over the place, and they are intertwined in the physical memory;
- There's no order in physical memory. A page in a low virtual memory address doesn't always end up in a low physical memory address. For example, the page that contains stack of process 1 is at the lower physical address, while the page with text segment is at the higher physical address;
- The pages are of **equal size**, so they are not aligned to a specific segment. For example, we see that one of the pages in process 2 has some part of the heap, and also some part of the data segment.

Now the ultimate question is, how does the processor know which physical address is corresponding to which virtual memory address in which process?! As to which process is running, it's related to *context switching*, a topic you'll learn in operating systems, so we'll skip it here and only focus on the first question.

4.3.3 Address Translation

The procedure of finding a physical memory address given a virtual address is called **address translation**. More formally, assume the virtual memory space is $\mathcal{V} = \{0, 1, \dots, N-1\}$ and the physical memory space $\mathcal{P} = \{0, 1, \dots, M-1\}$. Address translation can be defined as a function such that

$$\texttt{MAP} : \mathcal{V} \mapsto \mathcal{P} \cup \{\emptyset\} \tag{4.2}$$

Thus, given a virtual address a, we have $a' = \texttt{MAP}(a)$ where a' is the translated physical address. If $a' \in \mathcal{P}$, the virtual address can be successfully mapped to physical memory, and the data can be found there. If $a' = \emptyset$, the virtual address cannot be mapped to physical memory. This situation happens when, for example, we are trying to use a variable that's currently in a virtual page, but that page hasn't been loaded into physical memory. Thus, the first case is called a **page hit**, while the second **page fault**.

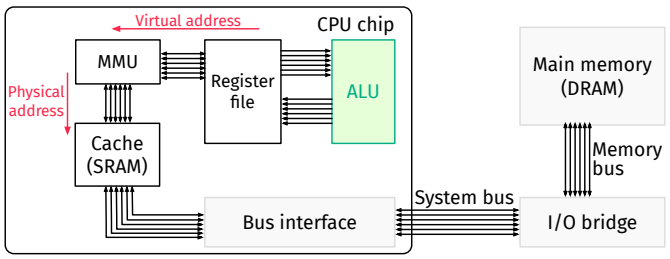


Figure 4.20: Adding memory management unit (MMU) to CPU chip.

4.3.3.1 Translation with Page Table

The translation from virtual memory address to physical address is performed by a special hardware called **memory management unit** (MMU). As shown in Figure 4.20, we add MMU to the CPU chip between the processor (register file and ALU) and SRAM cache. MMU receives a virtual address request issued by CPU, translates it to a physical address, and sends it to the cache. MMU's main job is to do the address translation to enable multiple tasks/processes share the main memory. It's managed and programmed by the operating system.

The address translation is nothing fancy—it's simply a one-to-one mapping by using a look-up table called **page table**. The page table is stored in the main memory, and looks like this:

Valid bit	Physical Page Number (PPN)
0	0x0000
1	0x0110
1	0xCC00
⋮	⋮

Each row in called a **page table entry**, and has two fields: valid bit, and **physical page number** (PPN). The index to each page table entry is called **virtual page number** (VPN). For example, if given a virtual address we know the VPN of it is 2, then by looking up the table above, we know that this virtual page is located at the address starting from `0xCC00` in the physical memory. The valid-bit indicates if the page is actually in the physical memory. If it's 1 then yes and we'll have a page hit; otherwise we'll have a page fault.

When there's a page fault, the main memory will pick a page called victim page, and evict it to the secondary storage, such as the hard drive. Then the page we need will be taken back to the main memory from the hard drive, and the address will be translated again by the MMU.

Here's a real-world example to help you understand. Assume you have multiple pages of a spreadsheet, where each page has 100 rows. What's the easiest way to locate to a specific row? You don't want to say, get me row 4,552 from the very first row. Instead, it'll be eaiser to say, get me row 52 on page 45, right? So you *index* to page 45, and count from the first row on *that* page to 52. Notice how we determine the location: the first two

digits 45 is the page number, and 52 is an *offset*. Bingo! That's exactly how we translate virtual address to physical address.

A virtual address of n bit can be split into two parts:

Bits `n-1..p`	Bits `p-1..0`
Virtual page number (VPN)	Virtual page offset (VPO)

Given a n-bit virtual address, MMU first splits it into two parts. The highest $n-p$ bits are used as the index to a row in the page table, and that's the virtual page number. If the valid bit is `1`, we have a page hit, then we retrieve the physical page number of that row. The virtual page offset is the lowest p bits, and they will simply be used as physical page offset (PPO) without change. So we attach PPO to PPN, and we have the physical address.

The contents in physical or virtual pages are simply bytes, so the page offset indicates which byte in that page. For example, if PPO $= 0$, we want the first byte of that page; if PPO $= 1600$, we want the 1601-th byte, and so on. Since we need p bits to represent the offset, and each byte has its own address, it is clear that given a page, there'll be 2^p bytes stored. Therefore, we call $P = 2^p$ the **page size**.

Quick Check 4.4

Assume the physical address has 32 bits, while the virtual address 64 bits. Based on different page sizes, determine the number of bits needed to represent VPN, VPO, PPN, and PPO.

P	VPN	VPO	PPN	PPO
1KB				
2KB				
4KB				
16KB				

Example 4.2

Now let's look at an example of translating virtual address to physical address. Assume:

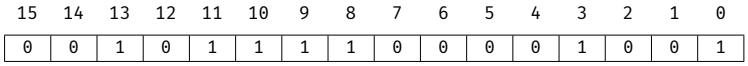
- The memory is byte addressable;
- Virtual addresses are 16 bits wide;
- Physical addresses are 14 bits wide;
- The page size is 1024 bytes.

In the following tables, all numbers are given in hexadecimal. The contents of the page table for the first 15 pages are as follows:

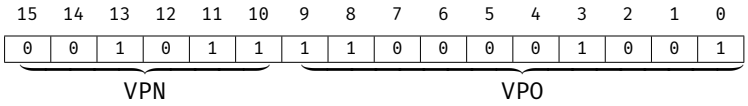
Page Table								
VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid
00	2	0	06	B	0	0A	3	0
01	5	1	07	D	1	0B	1	1
02	7	1	08	7	1	0C	0	1
03	9	0	09	C	0	0D	D	0
04	F	1	0A	3	0	0E	0	0

What's the physical address of virtual address `0x2F09`?

Solution: The first step is to write the virtual address in binary:

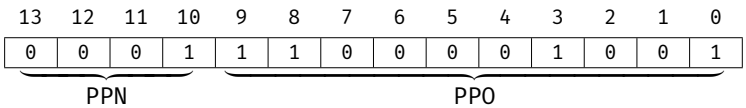


Then we need to decide various parameters for this virtual address. Because the page size is $1{,}024 = 2^{10}$ bytes, both VPO and PPO need 10 bits to represent. The virtual address is 16-bit, so the leading 6 bits will be used for VPN. Therefore, the virtual address is parsed as the following:



We convert VPN to hexadecimal and get VPN = `0x0B`. According to the page table, corresponding PPN for VPN of `0x0B` is 1, and since its valid bit is 1, we have page hit!

Physical addresses are 14 bits wide. Because VPO is identical to PPO, meaning it'll take 10 bits, we only need four bits for PPN. Therefore, we zero extend PPN to 4 bits: `0x0001`, and attach it with VPO:



This is the physical address in binary, so the last step is to simply convert this into hexadecimal, and we get physical address of `0x709`.

4.3.3.2 Accelerating Translation with TLB

Notice that the page tables are nothing special: it resides in the main memory just like any other data, such as our code, or other process data. Therefore, it'll be cached in L1 cache (and so L2, L3, *etc*) as well, which means it'll be evicted and replaced if the cache is full and we need to store other data in the cache. However, remember the special role of page table: it is used for translating addresses, which is almost needed all the time, so we certainly don't want it to be replaced too often. Otherwise the delay of retrieving page table entry would be too long.

Similar to the idea of caching memory data in a SRAM cache, we also add a small SRAM cache called **translation lookaside buffer** (TLB) on MMU, as in Figure 4.21. Essentially this TLB is also a set-associative cache; each line of this cache stores one page table entry.

We're fairly familiar with SRAM caches by now, so the indexing rule to a line is the same to TLB. Because page table entries can be determined by VPN, we separate VPN into two parts: TLB tag (TLBT), and TLB index (TLBI). These two parts will be used to index a page table entry (a line) in TLB:

Bits `n-1..p+t`	Bits `p+t-1..p`	Bits `p-1..0`
TLB tag	TLB index	Virtual page offset (VPO)

Virtual page number (VPN)

Here we use the middle t bits to indicate TLB set index, so in total we have $T = 2^t$ sets. Figure 4.22 shows the idea of TLB, which as you see, is just a SRAM cache where the cached data are page table entries.

Quick Check 4.5

Are the following statements true of false? Why?

1. If a page table entry can not be found in the TLB, then a page fault has occurred;
2. The virtual and physical page number must be the same size;
3. The virtual address space is limited by the amount of memory in the system;
4. The page table is accessed before the cache.

4.3.4 From Virtual Address to Data

In this section, we will look at a concrete example to see what actually happened during the entire address translation process. Let's start with something we're most familiar with. Say there's a statement in a C program we wrote:

```
1 char c = *ptr;  // ptr is a char pointer
```

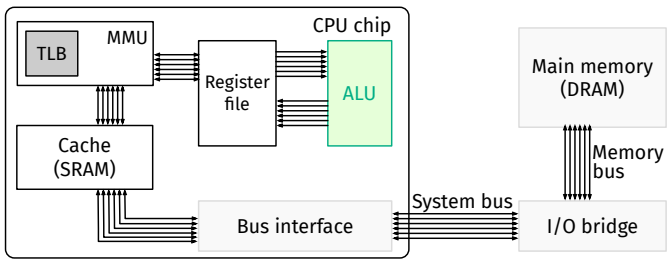


Figure 4.21: TLB as a cache for page table entries on MMU.

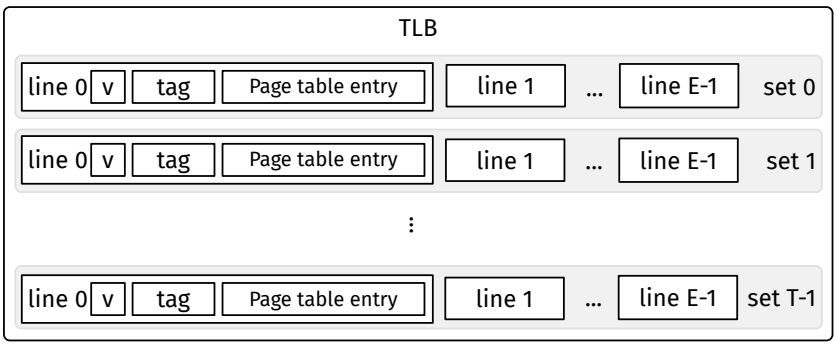


Figure 4.22: A translation lookaside buffer (TLB) is simply a SRAM cache that caches page table entries.

where `ptr` is a `char` pointer that points to a byte in the **virtual memory**. During compilation, this line will be translated into assembly:

```
1 LDURB  W0, [X1]
```

where we assume `X1` stores the value of `ptr`, which is a **virtual memory address**. From what we learned in Chapter 3, we know that during EX stage, the value of `X1` will be passed through ALU, and used as memory address for memory read transaction. Suppose the value stored in `X1` is `0x1DDE`. To retrieve the data stored at this virtual memory address, we need to do address translation.

Now assume the machine we run the above code has the following configurations

- The memory is byte addressable;
- Virtual addresses are 16 bits wide;
- Physical addresses are 13 bits wide;
- The page size is 512 bytes;
- The TLB is 8-way set associative with 16 total entries;
- The cache is 2-way set associative, with a 4 byte line size and 16 total lines.

At the time when we execute the `LDURB` instruction, the contents of the TLB, the page table for the first 32 pages, and the cache are as shown in Table 4.3

Step 1: Decide bit representations of virtual address.

Before we translate the address, we need to know which part of the virtual address is VPN, which part is VPO. This information is determined by the organization of the page table as well as TLB.

From the description, we know that page size is $512 = 2^9$ bytes, so we need 9 bits to represent page offsets, and therefore bits `[8..0]` is VPO. Since the virtual addresses are 16 bits wide, the leading seven bits `[15...9]` is VPN.

VPN consists of TLBI and TLBT. From the table, we see that TLB has only two sets, therefore one bit is enough to index to either. Thus,

Table 4.3: Contents of TLB, page table (first 32 pages), and the cache in the example. All numbers are **hexadecimal**.

TLB

Index	Tag	PPN	Valid
0	09	4	1
	12	2	1
	10	0	1
	08	5	1
	05	7	1
	13	1	0
	10	3	0
	18	3	0
1	04	1	0
	0C	1	0
	12	0	0
	08	1	0
	06	7	0
	03	1	0
	07	5	0
	02	2	0

Page Table

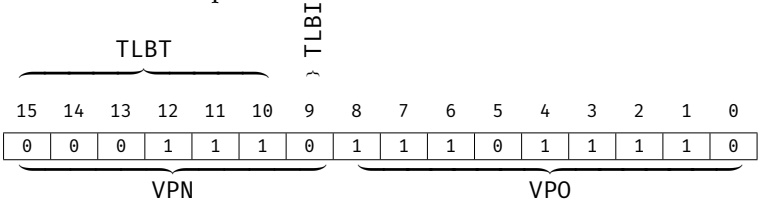
VPN	PPN	Valid	VPN	PPN	Valid
00	6	1	10	0	1
01	5	0	11	5	0
02	3	1	12	2	1
03	4	1	13	4	0
04	2	0	14	6	0
05	7	1	15	2	0
06	1	0	16	4	0
07	3	0	17	6	0
08	5	1	18	1	1
09	4	0	19	2	0
0A	3	0	1A	5	0
0B	2	0	1B	7	0
0C	5	0	1C	6	0
0D	6	0	1D	2	0
0E	1	1	1E	3	0
0F	0	0	1F	1	0

2-way Set Associative Cache

Index	Tag	Valid	0	1	2	3	Tag	Valid	0	1	2	3
0	19	1	99	11	23	11	00	0	99	11	23	11
1	15	0	4F	22	EC	11	2F	1	55	59	0B	41
2	1B	1	00	02	04	08	0B	1	01	03	05	07
3	06	0	84	06	B2	9C	12	0	84	06	B2	9C
4	07	0	43	6D	8F	09	05	0	43	6D	8F	09
5	0D	1	36	32	00	78	1E	1	A1	B2	C4	DE
6	11	0	A2	37	68	31	00	1	BB	77	33	00
7	16	1	11	C2	11	33	1E	1	00	C0	0F	00

in the virtual address, bit `[9]` is TLBI, while bits `[15...10]` is TLBT.

Step 2: Write out and parse the virtual address in binary.
Once the components of the virtual address have been identified, we need to write the address in binary, and recognize each component. In this example, we assume the virtual address sent by CPU is `0x1DDE`. Thus, we parse it as follows:



From this we have TLBT is `0x07`, TLBI `0x00`, VPN `0x0E`, and VPO `0x1DE`.

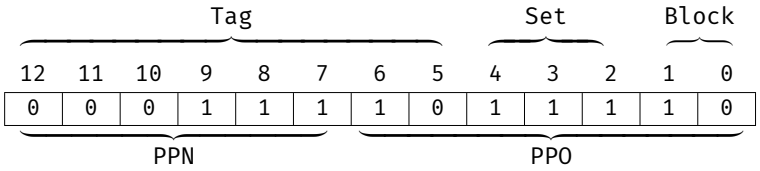
Step 3: Index into TLB.
Given the TLBI, we index into set 0 of TLB, and see if there's any line that has TLBT of `0x07`. Unfortunately there's no line's tag matches, so we have a TLB miss. Therefore, we'll have to use VPN to check

the page table. When there's a TLB miss, MMU will go to the main memory and fetch the page table entry given the VPN.

Based on Table 4.3, we see that the page table entry for VPN `0x0E` has a PPN of `0x1`. Fortunately, it's valid bit is 1, so we have a page hit, instead of a page fault. MMU will take this entry back, and translate it into physical memory address.

Step 4: Translate into physical address.
Now that we retrieved the PPN, MMU takes PPN and attach it with VPO to form the physical address, which is `0x11DE`. This is the real address in the physical memory where our variable `ptr` points to.

Step 5: Write out and parse the physical address in binary.
Once again, we need to determine the components of the physical address to utilize the cache. From the last step, we know that physical address is `0x11DE`. For cache, we need to determine the bits for the tag, set index, and block offset. Because the cache has $8 = 2^3$ sets, we need 3 bits for the set index. Each line has $4 = 2^2$ bytes, so we need 2 for the block offset. The rest of the $13 - (2 + 3) = 8$ bits will be used for tags. Thus, we write out the address in binary:



Therefore, the tag is `0x1E`, set index `0x7`, and block offset `0x2`.

Step 6: Visit cache.
Given the parsed result of the physical address, we first visit the cache. In set 7, we see that the second line has a tag `0x1E` which matches the one in our address, and its valid bit is 1, so we have a cache hit!! The block offset is `0x2`, so the byte we retrieved is `0xF`. This byte will be brought back to the CPU and stored in the register `W0`. Now, we know in our C program in the beginning, variable `c` will store `0xF`.

Note that if we had a cache miss, we'd have to go to the main memory, and evict the line with tag `0x1E` in set 7. This involves two operatioins: write the line in the cache back to the main memory (assuming it's write-allocate), and copy the line we want to the cache.

4.3.5 Summary

After all this mess, let's take a look at a big picture about how data are moved around inside our computers in Figure 4.23.

We write our code and compile them into executables. Those executables are stored on hard drive, and have the complete image of virtual mem-

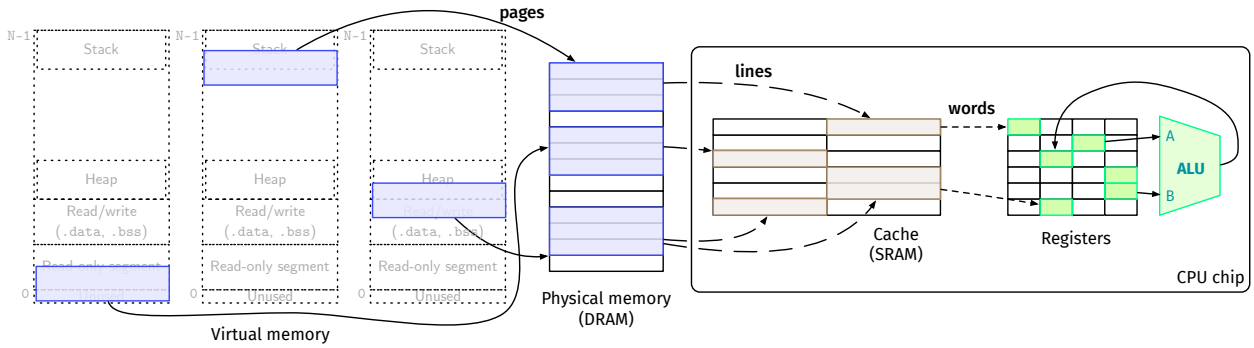


Figure 4.23: Physical memory retrieves and stores pages from the hard drive, and sends lines to the cache. The cache then sends words to the registers, where the data are requested by the ALU.

ory. From our perspective, each program has its own and identical virtual address spaces.

When we invoke a program, parts of that virtual memory will be carried into the physical memory as pages. Then some parts or data in that page will be carried into the cache as lines. When CPU requests a data and we have a cache hit, the data will be moved into registers as words (4 bytes), and loaded as operands to the ALU for calculation. This should give you a clearer picture of what's happening behind the scene.

As to the details on how exactly these things happened, it's way beyond our scope, and one course is not enough to talk about them in details. If you are interested in this, you might want to take system courses to study them in depth, and I encourage you to, because it's really fun!

4.4 Reference

Of course the contents in this chapter have been extremely simplified to make sure you focus on the most important concept, not distracted by too much details and technical discussions. For those who are interested to learn more in depth on ARM CPU chips, however, I recommend visiting the official website of ARM, specifically this one: `https://developer.arm.com/documentation/den0024/a/Caches`. You can see it includes contents such as cache, virtual memory, address translation, and so on. It has a lot more details, but you'll realize with the concept learned through this chapter as a foundation, it is not very difficult to hop on it and explore further.

Parallel Processors 5

What we have learned so far has focused on single core machines, meaning there's only one CPU for the computer. That's almost unimaginable these days, since all computers have multiple cores to speed up the execution of tons of applications we're running on our computers.

There are a few ways to categorize different types of parallel processors. Based on if it's running multiple instructions or sending multiple data streams at a time, we categorize them as instruction and data level parallelism, as in Table 5.1, with examples of processors that fall in that category.

5.1 Instruction Level Parallelism (ILP)

We'll start with instruction level parallelism first, which can extend what we have learned in Chapter 3.

Recall that in Chapter 3, we learned that pipelining can greatly help the program throughput, since we can execute multiple instructions at the same time. However, as we deepen the pipeline, the overhead of register use is getting worse. So what if we still use the current pipeline, but create multiple of them? That's the idea of instruction level parallelism.

For example, if we have two pipelines, we can issue two instructions at a time, and at the peak there'll be 10 instructions in total, which doubles the instructions processed with only one pipeline. We call this **multiple issue**, meaning "issue multiple instructions at each clock cycle". In general, multiple issue can be done from two aspects, either from software or hardware.

5.1.1 Static Multiple Issue

Software-wise it is called **static multiple issue**, since it's managed by the compiler almost entirely. Compiler groups instructions to be issued together, and packages them into "issue packets". An **issue packet** contains multiple concurrent instructions. It is sometimes called Very Long Instruction Word (VLIW) because, for example, two ARM instructions together occupies 64 bits, and four would make it 128 bits.

Table 5.1: Categorization of parallel processors.

	Static: Discovered at compile time	Dynamic: Discovered at runtime
Instruction level	VLIW	Superscalar
Data level	Vector	GPU

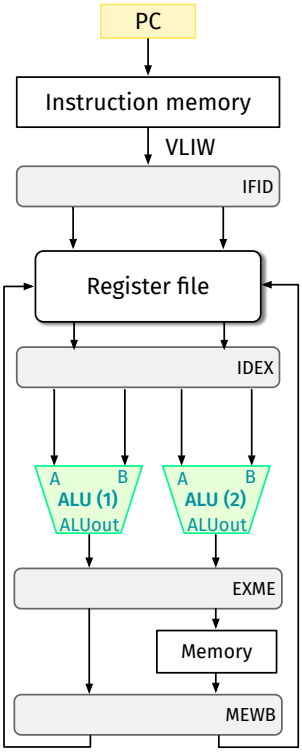


Figure 5.1: A static dual issue model, where ALU(1) is used for arithmetics, while ALU(2) for load and store instructions.

Let's look at a very simple static multiple issue, called **static dual issue**. It's not difficult to infer from the name that in this model, each time the processor issues two instructions together. We show a simplified diagram in Figure 5.1 for this model.

In this model, the most obvious thing we notice is there are two ALUs. The processor retrieves an issue packet that contains two instructions from instruction memory each time. The two instructions will share everything, including register file, pipeline registers, *etc*, but will use two separate ALUs. Another thing we notice is only the route on the right has access to memory in the ME stage. Typically, for a model as in Figure 5.1, we use ALU(2) exclusively for load and store instructions, while ALU(1) for arithmetics and branching. Since instructions such as `ADD` do not need to access memory, we let it bypass the memory to reduce delay.

5.1.1.1 Scheduling

As expected, static multiple issue will still have problems with data or control hazards. The compiler therefore has to rearrange the instructions to avoid them. One strategy is called **scheduling**, where if the hazard cannot be avoided, the compiler needs to manually insert `NOP`s, or even to modify the original code as necessary.

The code below is a segment to add a scalar (stored in `X21`) to every element in an array of double words, starting at address `X20`.

```
1 Loop: LDUR X0, [X20]     // X0 = array element
2       ADD  X0,  X0, X21  // add scalar in X21
3       STUR X0, [X20]     // store result back
4       SUB  X20, X20, 8   // decrement pointer
5       CMP  X20, X22      // branch if X20 > X22
6       B.GT Loop
```

To schedule this segment in a static dual issue model, for each cycle we need to issue two instructions together: one for arithmetic, and the other for memory access. Let's categorize all instructions in the code segment first:

Arithmetic/Branching	Load/Store
`ADD X0, X0, X21`	`LDUR X0,[X20]`
`SUB X20, X20, 8`	`STUR X0,[X20]`
`CMP X20, X22`	
`B.GT Loop`	

For the first pair, there's nothing we can/should do without loading in the element first. Therefore, for the first packet, we put `NOP` on the left to let `LDUR` load the element first. In cycle 2, we cannot add `X21` to `X0` yet, because recall that the data read from `LDUR` will not be available until cycle 4. Instead, `X20` has already been read from the register in cycle 2 and used

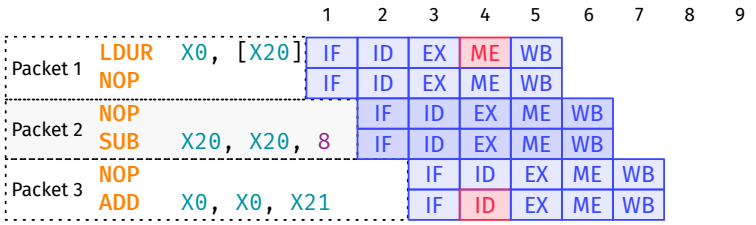


Figure 5.2: The pipeline diagram showing the first three packets for the static dual issue model. The LDUR's ME stage is aligned with ADD's ID stage, and both are highlighted.

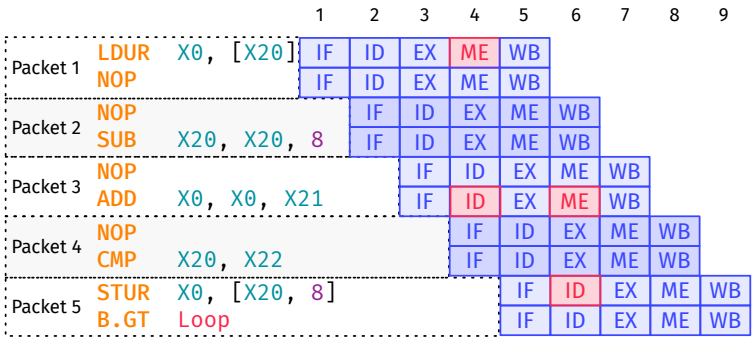


Figure 5.3: The pipeline diagram showing one complete iterations of the example.

by LDUR for calculating address, so we can let the SUB instruction execute first; there's no data dependency between ADD and SUB anyway. In cycle 3, we issue ADD and NOP combo, so that in cycle 4 ADD's ID stage can be aligned with LDUR's ME stage. We show a pipeline diagram in Figure 5.2 for the first three packets, and the scheduled code below:

Cycle	Arithmetic/Branching	Load/Store
1	`NOP`	`LDUR X0,[X20]`
2	`SUB X20, X20, 8`	`NOP`
3	`ADD X0, X0, X21`	`NOP`

We cannot issue STUR instruction in either cycle 3 or 4, because the new value of X0 won't be available until cycle 5. Lastly, at cycle 5, we are going to store the new value of X0 back to memory. What's critical here is to notice, originally the instruction was `STUR X0,[X20]` , because it follows ADD instruction closely. After rearranging the instructions, X20 has been changed to `X20 - 8`, so we have to use original address, which is X20 (new value) `+ 8`:

Cycle	Arithmetic/Branching	Load/Store
1	`NOP`	`LDUR X0,[X20]`
2	`SUB X20, X20, 8`	`NOP`
3	`ADD X0, X0, X21`	`NOP`
4	`CMP X20, X22`	`NOP`
5	`B.GT Loop`	`STUR X0, [X20,8]`

5.1.1.2 Loop Unrolling

Another common strategy used by compilers, especially when there's a loop over an array in the code, is **loop unrolling**. When we deal with loops, what we usually do is write one iteration at a time, and in each iteration we use the same registers. Loop unrolling is to expose multiple iterations at a time and use multiple registers. For example, in the code listing in the previous section, we load each array element to `X0`, and this is the same for every iteration. If we're using loop unrolling, however, we might load four elements at a time, and put them in registers `X0` to `X3`. A similar arrangement is shown below:

Cycle	Arithmetic/Branching	Load/Store
1	`SUB X20, X20, 32`	`LDUR X0, [X20]`
2	`NOP`	`LDUR X1, [X20, 24]`
3	`ADD X0, X0, X21`	`LDUR X2, [X20, 16]`
4	`ADD X1, X1, X21`	`LDUR X3, [X20, 8]`
5	`ADD X2, X2, X21`	`STUR X0, [X20, 32]`
6	`ADD X3, X3, X21`	`STUR X1, [X20, 24]`
7	`CMP X20, X22`	`STUR X2, [X20, 16]`
8	`B.GT Loop`	`STUR X3, [X20, 8]`

Notice for `X0`, `LDUR` and `STUR` use different offsets in cycle 1 and 5. This is because in cycle 1, we also substracted 32 bytes from `X20`, so in cycle 5 we need to use the old value for the memory address, which can be calculated as `X20+32`. There's no difference for other elements, because at the point where they read `X20` in `LDUR` instructions, the value of `X20` has already been updated, and will stay the same until next `SUB` instruction.

5.1.2 Dynamic Multiple Issue

Only static multiple issue is not enough, as not all stalls are predictable, such as cache miss. Additionally, not all instructions can be rearranged; branchings, for example, might depend on the value of a register during *runtime*, which cannot be determined by the compiler during *compilation time*. Using hardware to manage stalls, therefore, is also necessary. In this case it's called **dynamic multiple issue**. Processors using dynamic multiple-issue strategy are also called **superscalars**. We show the typical architecture of a superscalar in Figure 5.4.

In this architecture, we have four major parts:

- **Instruction fetch and decoding unit:**
  This unit is very similar to the IF and ID stage in our pipeline; it retrieves VLIW and distribute them into different functional units for specific types of calculation;

- **Reservation station:**
  Before sending the decoded instructions—including operands and operators—to funtional units to execute, we buffer them into a reservation station first. This is to make sure all the operands used by an instruction is ready to avoid data hazard;

- **Functional units:**
  Once it's ready, the functional units will retrieve operands and operators from its reservation station, and do the calculation correspondingly. We separate the functional units to different parts, just like what we did in static multiple issue example, where we have one ALU for integer calculation, and another for address calculation for `LDUR` and `STUR`. Notice we use a different unit to calculate floating point numbers, since the typical ALU we have learned cannot deal with floating points;

- **Commit unit:**
  The results of functional units will be sent to the commit unit, where they will first be put into a **reorder buffer**. Again, the reason of using this is to avoid data hazard. When it's ready, *i.e.*, no more data hazard, the rearrange the results and write them back to the register file.

As we see from the description above, the superscalar pipelines have generally three stages:

- **In-order issue**, where the VLIWs are issued in the order decided by the compiler;
- **Out-of-order execute**, where each funtional units execute its instructions at its own pace to avoid data hazards;
- **In-order commit**, where after gathering all the results from functional units, the commit unit re-order them so that the state of the program is still the same as the order of the instructions issued. So in the end, for us programmers, it looks like the instructions are fetched and executed in the order they were written in assembly.

To save you a Wikipedia search...

In computer engineering, an execution unit (E-unit or EU) is a part of the central processing unit (CPU) that performs the operations and calculations as instructed by the computer program. It may have its own internal control sequence unit (not to be confused with the CPU's main control unit), some registers, and other internal units such as an arithmetic logic unit (ALU), address generation unit (AGU), floating-point unit (FPU), load-store unit (LSU), branch execution unit (BEU) or some smaller and more specific components.
It is common for modern CPUs to have multiple parallel functional units within its execution units, which is referred to as superscalar design. The simplest arrangement is to use a single bus manager unit to manage the memory interface, and the others to perform calculations. Additionally, modern CPUs' execution units are usually pipelined. (From Wikipedia page on *Execution Unit*.)

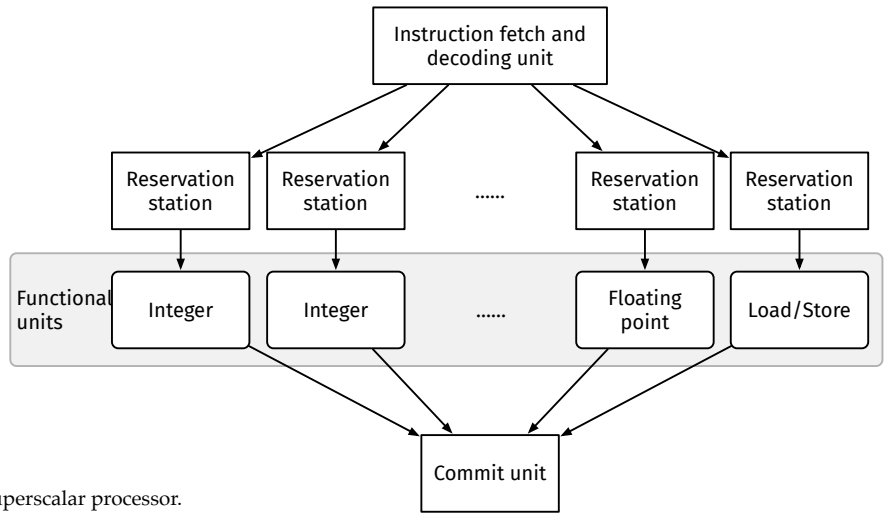


Figure 5.4: A typical architecture of a superscalar processor.

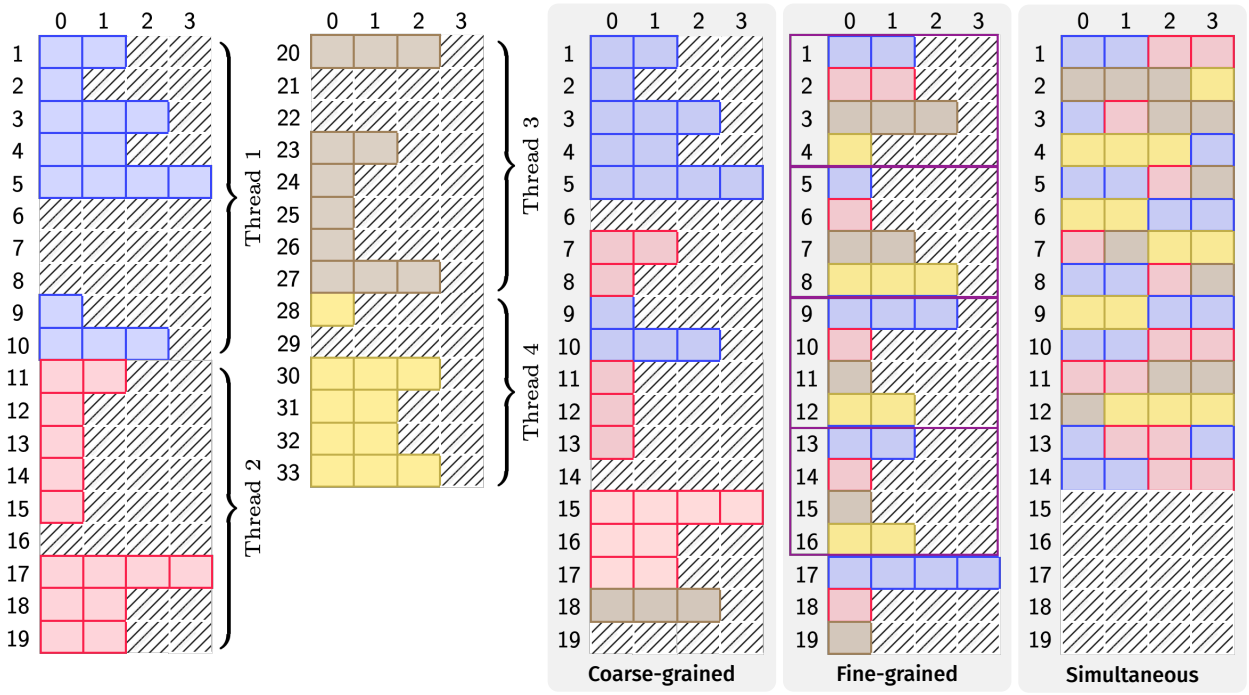


Figure 5.5: An example showing three approaches to hardware multithreading. Each row is a clock cycle, and each column is a processor core. If a cell is colored, the processor issues an instruction to that core at that clock cycle; otherwise it's idle.

5.1.2.1 Hardware Multithreading

All the programs we have written so far are single-threading programs. We wouldn't go into details about threads and processes here, but briefly, a process can have multiple threads running at the same time, and they share the virtual memory address space. Each of the threads has its own PC, register state, and stack area. The idea of hardware multithreading is to reduce delay by switching to another thread *within* a process when one thread has been stalled.

There are three styles of hardware multithreading:

- **Coarse-grained multithreading**, which only switches on long stalls (*e.g.*, L2- or L3-cache miss). It simplifies hardware design, but doesn't hide short stalls such as data hazards;
- **Fine-grained multithreading**, which switches threads after each cycle;
- **Simultaneous multithreading**, which tries to fill in all the cores ot make them busy.

Let's look at an example in Figure 5.5, where we run our process with four threads on a four-core superscalar. Without multithreading (as in the left side of the figure), these threads have to execute sequentially, which takes 33 cycles to finish the entire program. Each row is a cycle, where we issue VLIW on the cores; some VLIWs only need one or two, while others need more. The gaps of cycles within each thread can be caused by stalls, such as cache miss or data hazards. The three approaches of hardware multithreading are shown on the right side.

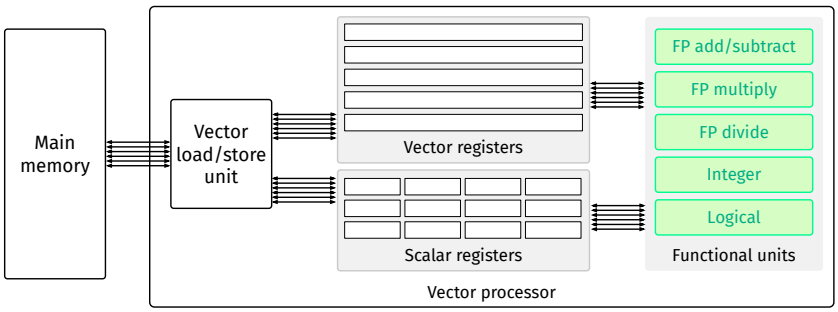


Figure 5.6: A typical architecture for vector processors.

5.2 Data Level Parallelism (DLP)

Instruction level parallelism can improve the system performance to an extent, but has its own limits as well. As you can imagine, the hardware design and implementation of deeper pipelines and all the components are already a challenge. Wiht exploiting ILP comes increased compiler and circuitry complexity.

Another perspective to improve performance is to exploit data level parallelism (DLP). Similar to ILP, we also have two flavors in DLP: static and dynamic.

5.2.1 Static: Vector Processors

CPU that implements an instruction set that operates on 1D arrays are called **vectors**; Just like the vectors from math, here a vector contains multiple data elements, and a vector processor can process a bunch of them at the same time.

For the code segment below:

```
for (int i = 0; i < n; i ++) {
    c[i] = a[i] + b[i];
}
```

we're trying to add arrays (vectors) of length `n` together. When $n \to \infty$, the number of operations needed on a single core machine increases linearly, since each time only one operation of addition is possible. Using a vector machine, however, the processor will possibly load all elements of `a` and `b` together, and perform addition on all elements, which greatly reduces the operation needed.

We show a typical architecture for vector processors in Figure 5.6. There are several key components in this architecture:

- **Vector registers:**
  there are typically 8–32 vector registers, each of which stores 64 elements of 64–128 bits. Therefore, if the array we're operating on has 32 elements, and a vector register stores 64 elements, we only need to load the entire array into one vector register. This already reduces

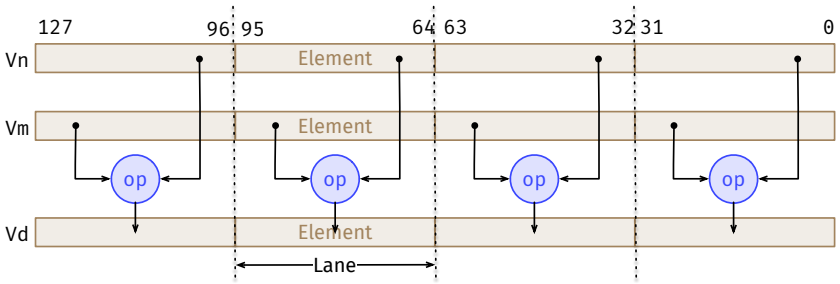


Figure 5.7: Operations on vector registers in ARM NEON processor: `Vd = Vn op Vm`.

time spent on load and store. The vector registers by default store double-precision real numbers, instead of integers;

- **Scalar registers:**
  In addition to vector registers, we also need scalar registers that can store only one 64- to 128-bit number, just like our `X0`, `X1`, *etc*. Not all operations have to be vectors; using vector registers to process scalar data is very much overkill;

- **Functional units:**
  The data loaded into scalar and vector registers will be sent to different functional units to perform calculation. Typically there are four to eight units that can deal with different operations. In the figure we have three floating point, one integer, and one logical operations;

- **Vector load/store unit:**
  Lastly, we have vector load/store unit to transfer data between the main memory and the vector and scalar registers.

So why do we say it's *static* DLP? The term "static" here has the same meaning as in ILP—the parallelism is discovered and utilized during *compilation time*. In fact, many processors have direct instructions to operate on arrays as well. For example, ARM NEON processor is a vector processor that has 32 vector registers of 128 bits, and has a specific set of SIMD instructions to process vectors. [1]

1: We'll explain what SIMD means in following sections.

Figure 5.7 shows how ARM NEON operates on vector registers. In this model, each vector register can hold four **elements**. The elements of two input vector registers at the same position will be calculated according to specific instruction operator, and sent the output to the destination `Vd`. The two operand elements and the destination element form a **lane**.

The following code is a classical example, usually called **DAXPY**, which is to calculate linear transformation $\mathbf{y} = a \times \mathbf{x} + \mathbf{y}$: [2]

2: D stands for double-precision, and P standands for the plus sign. Both **x** and **y** are vectors, while a is a scalar.

```
void daxpy(double* x, double* y, double a, int n) {
    for (int i = 0; i < n; i ++) {
        y[i] = a * x[i] + y[i];
    }
}
```

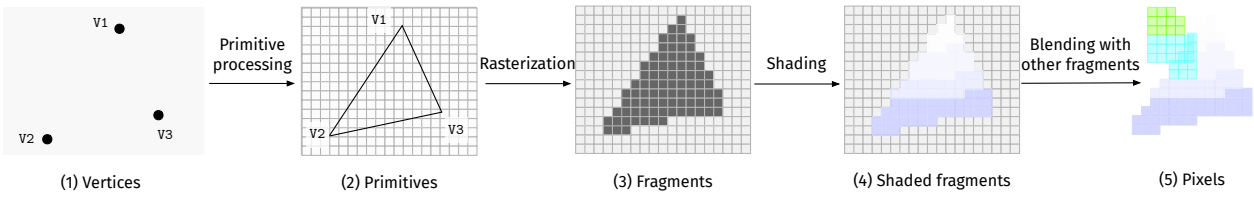


Figure 5.8: A streaming multiprocessor containing eight streaming processors.

Using conventional ARM instructions, the assembly will have to load `x[i]` and `y[i]` first, calculate the transformation, and store the result back. If we are using the NEON processor as in Figure 5.7, we can operate on four elements at a time, which greatly speeds up the processing. We will not show detailed assembly program for NEON processor here, but if you're interested, feel free to ask for more reference.

5.2.2 Dynamic: Graphics Processor Units (GPU)

For dynamic DLP processors, one of the most famous example is GPU, which was intended for graphics processing originally but has been used extensively for general purpose calculation these days. Before talking about the architecture of GPUs, let's briefly discuss graphics processing pipeline first. After all, *that* is the reason why GPUs exist, and it's especially beneficial to understand the architecture.

5.2.2.1 Graphics Processing Pipeline

In graphics processing, the most important and possibly most demanding task is to render a 3D image to display, and even make an animation. The process/pipeline of generating a graph we see on the screen is shown in Figure 5.8, where we generate a triangle. In fact, generating a triangle is so simple yet so ubiquitous in graphics processing, because all the images can be splitted into small triangles (usually called triangulation). Sort of like approximation in your calculus class.

Briefly, we start with vertices that "project" any 3D object to a 2D plane with the coordinate (x, y) of each of them. Next step is to generate shapes based on these vertices, called primitives. Primitives are then rasterized into "pixel fragments", to decide which pixel needs to be shaded, and then apply shading to these pixels to give them colors. After blending all fragments, we have the final image as pixels shown on the display. During this time, different textures can be applied to all intermediate componenets such as primitives and fragments. For example, you want the triangle to look like it's cut out of a tree, then just apply a wood texture to the fragment.

An animation consists of a large amount of frames where each frame is a static pixel image generated from the process above. To make the animation run smooth, the frame rate should be no less than 60 frames per second, meaning for each second, we need to at least render 60 3D images,

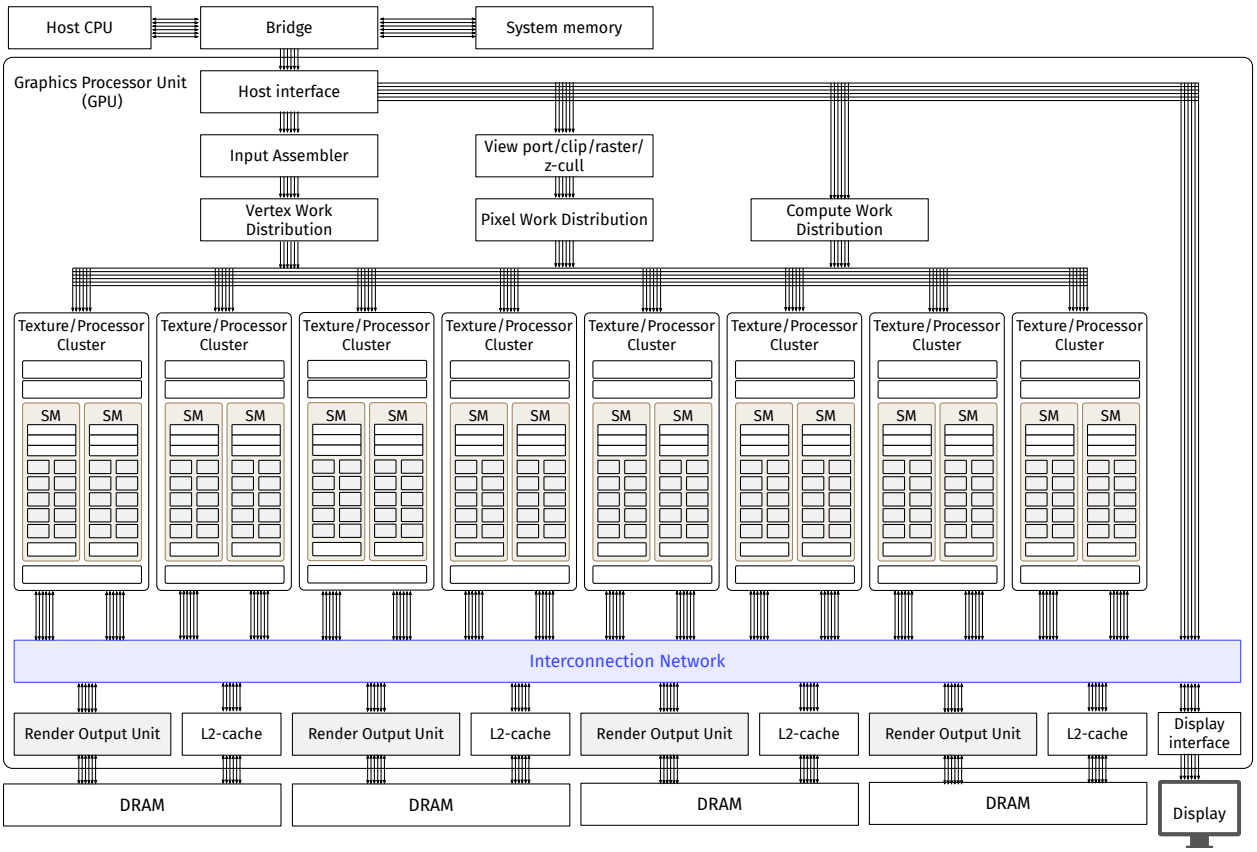


Figure 5.9: The architecture of GEForce 8800 GTX by NVIDIA.

from calculation to the actual display. Sixty images is really a lot considering all the factors we need to calculate. To make all these calculations happen in a few milliseconds absolutely requires heavy data parallelism.

5.2.2.2 Overview GPU Architecture

GPUs in the early days are designed specifically to accelerate this graphics processing pipeline. We will use GEForce 8800 GTX as an example, as in Figure 5.9. Following the steps described in Figure 5.8, the first thing we need to do is to generate primitives. The host CPU sends out instructions to the GPU first, and the input assembler in GPU receives the 3D representation as a collection of geometric primitives such as points, lines, and vertices. All the different parts of the image will be distributed to Texture/Processor Clusters (TPC) for calculation. The results from multiple TPCs will be collected and arranged in the interconnection network. Several Render Output Units (ROP) render the calculation result to the actual pixel image, and transfer it to the display interface, which is the image we see on our screen.

Texture/Processor Cluster: As we see in Figure 5.9, the calculation work will be distributed into a couple of Texture/Processor Clusters (TPC).

The word "texture" comes from its origin of graphics processing apparently. We show a diagram of TPC in Figure 5.10. The two streaming multiprocessors are controlled by a Streaming Multiprocessor Controller (SMC) and a Geometry Controller (GC). They also share a texture unit where a L1-cache is used.

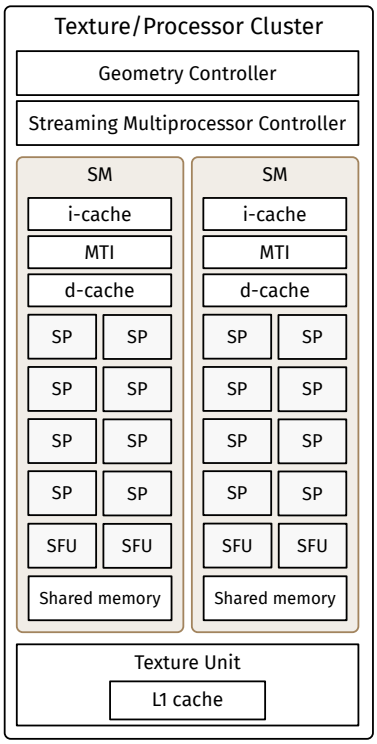


Figure 5.10: Texture/Processor Cluster.

Streaming Multiprocessors: In Tesla-based models, each TPC typically contains two **nodes**, or **streaming multiprocessors** (SM). Each of them contains eight multithreaded single-precision floating point and integer processing units, which used to be called **streaming processors** (SP) but are now also called **cores**.

Figure 5.11 shows a typical structure of streaming multiprocessors. In addition to eight streaming processors, there are also two caches, d-cache and i-cache, that buffer data and instructions. Multithreading issue unit is used to issue multiple instructions and implement hardware multithreading. Special function units are used to perform special calculations such as `sin`, `cosine`, reciprocal, and square root, to achieve faster approximation calculation with less precision. Lastly, shared memory is used to coordinate data sharing among threads.

5.2.2.3 Performance Measurement

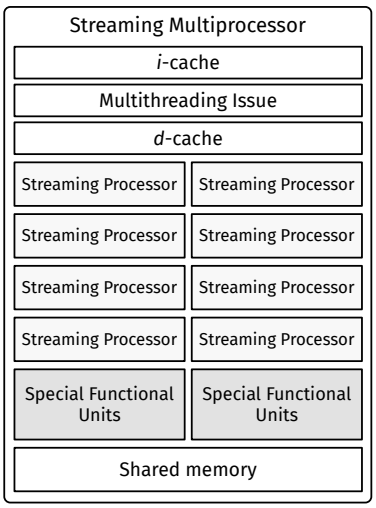


Figure 5.11: A streaming multiprocessor containing eight streaming processors.

Let's talk about numbers now to see how powerful GPUs are. We define **peak performance** as the performance when all processors are kept busy all of the time. Since GPUs can run billions of instructions per second, we specify a unit used to measure peak performance, called **GLOPS**: billions of floating-point operations per second.

Take GEForce 8800 GTX as an example. The machine has 16 streaming multiprocessors, 8 cores per streaming multiprocessor, with a clock rate of 1.36GHz. Assume we run two FLOPs—floating point multiplication and addition—in each clock cycle for each core. The peak performance is calculated as follows:

Table 5.2: Flynn's Taxonomy of parallel processors.

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: Vector architecture
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

$$
\begin{aligned}
\text{Peak performance} &= \mathbf{16}\ \text{SMs} \times (\mathbf{8}\ \text{cores/SM}) \\
&\quad \times \mathbf{2}\ \text{FLOPS/instruction (add and multiply) /core} \\
&\quad \times \mathbf{1}\ \text{instruction/clock cycle} \\
&\quad \times \mathbf{1.35} \times \mathbf{10^9}\ \text{clock cycles/second} \\
&= 16 \times 8 \times 2 \times 1.35 \times 10^9 \text{FLOPs/second} \\
&= 345.6\ \text{GFLOPs / second.}
\end{aligned}
$$

You read that right: all 345.6 **billions** of floating point calculations can be done within one second!

5.3 Flynn's Taxonomy

Another classic categorization of parallel processors is from 1960s, as in Table 5.2, called Flynn's Taxonomy, thanks to Michael Flynn. Notice, however, GPU doesn't fit nicely into any category here, although sometimes with careful programming GPUs could create an illusion of MIMD.

5.4 Interconnection Networks

When one computation job is split and distributed into multiple processors, the communication and coordination between the processors becomes more important than ever. You must've noticed from previous sections that a common structure is to utilize an interconnection network to connect all processors. This network becomes a "channel" that allow processors to exchange intermediate results, share local variables, and so on.

5.4.1 Graph Representation

A general organization of a multiprocessor is shown in Figure 5.12, where each processor-cache-memory tuple is called a **processor-memory node**. The interconnection network connects all nodes to pass data. In fact, if you treat each node as a separate computer, this organization can also be regarded as a computer cluster.

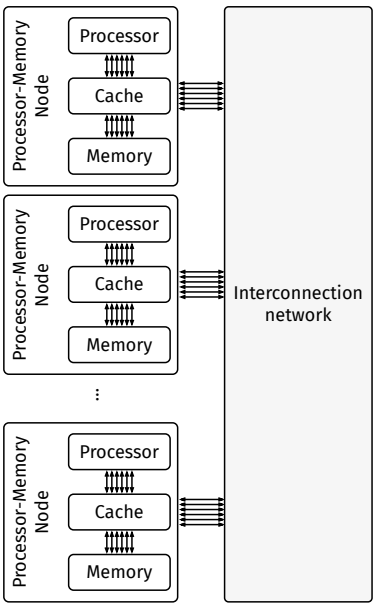


Figure 5.12: A typical organization of a multiprocessor.

3: In fact, this type of network is called switch-based interconnection network. There are other types as well.

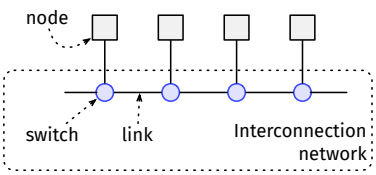


Figure 5.13: Abstract diagram of an interconnection network.

One major challenge in interconnection networks is to find an optimal structure that can transfer data efficiently. We make an abstract presentation of the network using the diagram in Figure 5.13. We use squares to represent processor-memory nodes, and each node is connected to a switch. [3] Switches are then connected through "links"—**bidirectional** buses that transfer data.

5.4.2 Performance Measurement

There are many different structures of interconnection network, so to choose one that yields the optimal efficiency is important. By efficiency, we refer to **bandwidth**: the number of bytes of data that can be transferred per second.

We introduce two popular measurements for evaluating different structures of interconnection networks: total network bandwidth and bisection bandwidth.

- **Total network bandwidth:** This measures the peak bandwidth, meaning when all the links have been used, how many bytes of data will be transferred per second. It's simply the bandwidth of each link multiplied by the number of links;

- **Bisection bandwidth:** Another more important measurement is bisection bandwidth. Assume we are going to divide a set of processors in two equal partitions (or at most the difference of one processor), which can lead to many different partitions. How much link bandwidth is available for data flow from one partition to the other? Take the smallest (worst) value over all possible partitions, and we get the bisection bandwidth. It can be calculated as follows:
  - Dividing the machine into two halves;
  - Sum the bandwidth of the links that cross that dividing line.

Let's look at some common topologies and see how to calculate their total network and bisection bandwidth.

5.4.3 Common Topologies

In the following, we let P denote the number of nodes (and therefore switches), and B the bandwidth on each link.

Ring (*k*-ary 1-cube, or wrapped chain)

The simplest structure is a **ring**, as shown in Figure 5.14, where each node has only one link. [4] It's not difficult to calculate the total bandwidth $P \cdot B$, and the bisection bandwidth $2 \cdot B$.

2D Meshes

The ring structure is simple, but has awful bisection bandwidth. Additionally, the all the nodes are connected, so if one node wants to send data to another that is far, the data has to be transferred to other nodes multiple times. **2D mesh** is a structure that addresses this problem.

We show a 2D mesh in Figure 5.15. For easier calculation, let's assume there are in total $P = K \cdot K$ processors. Since for a $K \cdot K$ 2D mesh there are in total $2 \cdot K(K-1)$ links, the total network bandwidth is $2 \cdot K(K-1) \cdot B$. To calculate bisection bandwidth, we just need to cut the grids in the middle, then we'll see that the number

4: Each node has technically two links, but you'll count all links twice in that case, so you need to divide that by 2.

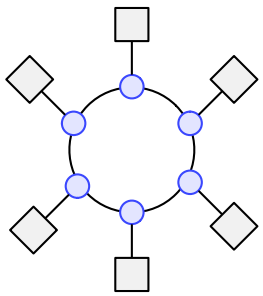


Figure 5.14: Ring structure.

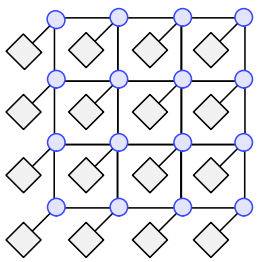


Figure 5.15: 2D mesh structure.

of links that cross through the two halves is K, and therefore the bisection bandwidth is $K \cdot B$.

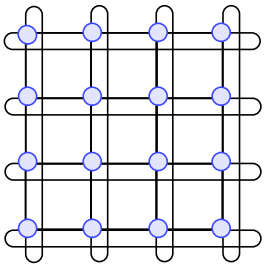


Figure 5.16: 2D torus structure. We removed the nodes to make the diagram clearer.

2D Torus (*k*-ary 2-cube)

You probably have noticed that 2D mesh also didn't connect nodes on the two sides. A **2D torus** structure is simply a 2D mesh that connects each row and each column's first and last node, as shown in Figure 5.16. In this case, each node has two links, so the total bandwidth is $2 \cdot P \cdot B$, and the bisection bandwidth is $\frac{1}{2}P \cdot B$.

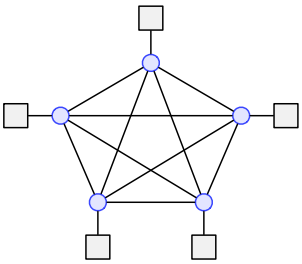


Figure 5.17: A fully connected structure.

Fully Connected Network

As the name suggests, a **fully connected network** is one where all the nodes can pass data to each other. From your discrete math class, you learned that a fully connected graph of P vertices has $\frac{1}{2}P \cdot (P-1)$ edges, so the total bandwidth is $\frac{1}{2}P \cdot (P-1) \cdot B$. The bisection bandwidth is $\left(\frac{P}{2}\right)^2 \cdot B$.

A summarization of common topologies and their performance can be found in Table 5.3. Of course there are many more different structures such as hypercubes, trees, fat trees, *etc*, but it's beyond the scope of our course. If you are interested in this topic, courses such as *Parallel Computing* would be great choices.

Table 5.3: Summary of common topologies and their performance, assuming there are $P = K^2$ nodes, and the bandwidth for each link is B.

Structure	Total bandwidth	Bisection bandwidth
Ring	$P \cdot B$	$2 \cdot B$
2D mesh	$2 \cdot K \cdot (K-1) \cdot B$	$K \cdot B$
2D torus	$2 \cdot P \cdot B$	$\frac{1}{2} \cdot P \cdot B$
Fully connected	$\frac{1}{2} \cdot P \cdot (P-1) \cdot B$	$\left(\frac{P}{2}\right)^2 \cdot B$

Appendix

C Language in Action | A

This chapter is served as a quick-and-dirty introduction to get started with C language programs.

A.1 Compile, Link, and Execute

Our C program should be stored as a source file, with extension `.c`. Open terminal, and `cd` to the directory where the source code is defined. The first step is to use `gcc` to compile the code:

```
$ gcc hello.c -c
```

This command will let `gcc` compile our source code `hello.c`, and generate an object file `hello.o`.

Next step is to link all the object files. Since in this example we only have one object file, we can simply use

```
$ gcc hello.o
```

which will generate an executable file, whose name is `a.out` by default, if there's no any error in our code. If there are multiple object files generated from different source files, say `hello.o`, `test.o`, and `demo.o`, the command line would be

```
$ gcc hello.o test.o demo.o
```

Then we can run the executable:

```
$ ./a.out
```

Note that `./` in the beginning is necessary.

If we want to generate an executable with another cute name instead of the default `a.out`, we need to use flags. For example, we want to name our executable as `cute`, then we should do:

```
$ gcc hello.o -o cute
```

To run this, type the following in the terminal:

```
$ ./cute
```

Note that the output file name (`cute` in this example) has to follow `-o`, but `-o output_name` can be anywhere after the command `gcc`. Remember, the values of the flags and the flags themselves always go together as a pair.

Usually the following steps can be simplified to just one command:

```
$ gcc hello.c
```

which will compile (without generating an object file explicitly), link, and generate an executable. For the rest of the course, feel free to use this command to speed up your workflow, but for this lab, you have to know how to use the flags to do the compilation and link separately. This will help you later when we write assembly programs.

A.1.1 Flags

You can have many flags attached to the command `gcc`. You can view them using the `--help` flag. This is a flag without any values:

```
$ gcc --help
```

Some commonly used flags are:

- `-Wall`: short for *warning all*, which will show all the warnings (they are not errors);
- `-g`: use this if we want to debug our code using `gdb`. More on this later;
- `-O`: capital O, for optimizing our code. It doesn't modify our source code; instead, it only generates more efficient executable file.

So, to generate an efficient C executable called `too_cute` that can be debugged, the command line looks like:

```
$ gcc -O -o too_cute -g hello.c
```

A.2 Debugging C Programs

We will use `gdb` to debug C programs. Later in assembly we will also use `gdb`, but that's a different version.

A.2.1 Installation

Generally Unix distributions come with `gdb`, but we can also install it using the following commands from terminal:

```
$ sudo apt-get update
$ sudo apt-get install gdb
```

For more details refer to http://www.gdbtutorial.com/tutorial/how-install-gdb.

If you're using macOS with M1 chip, you can use `lldb` instead of `gdb`, because the latter doesn't support M1 chip yet. If you're using macOS with Intel chip, you might want to follow the instruction provided at https://gist.github.com/danisfermi. Using the virtual machine, everything is good to go.

A.2.2 Debugging

In general we use the following command to compile our code:

```
$ gcc prog.cpp -o a.out
```

Now to enable our program to run with the debugger we need to put `-g` option So the command becomes:

```
$ gcc -g prog.cpp -o a.out
```

We can start the debugger on your program by one of the following two ways. We can type the following command

```
$ gdb a.out
```

We can also enter the debugger first by using

```
$ gdb
```

and then type the name of the executable `a.out`.

Let's get familiar with a few commands in `gdb` for running our code. Try some of these out, but note that some won't work yet because they rely on other `gdb` commands being run first.

```
# ask gdb to tell you about the named gdb command
help command

# run your code (and add command line arguments if any)
run [arg1 ... argn]
```

Once we gave the `run` command, you'll notice that our program has been executed from the start to the end. This looks useless because it's the same as we run our program from terminal directly. The advantage of a debugger is that we can set a **breakpoint** somewhere in our code, so that the program will pause there and so we can see what's going on in the program.

A.2.2.1 `break`

We need the debugger to pause at certain point in the code so that we can investigate the program. To set a breakpoint we will use the command `break`.

```
break prog.c:12
```

In this example we are setting a breakpoint in file `prog.c` at line number 12. If we are interested in a particular function we can set break point at that function and the debugger will pause every time the function is called:

```
break function_name
```

Once you have set breakpoints, you can give the `run` command, and you'll see this time the program is paused at the breakpoints.

A.2.2.2 `continue`

To move on to next break point we can use the command `continue`, or simply `c`.

A.2.2.3 `step` and `next`

To proceed by single-step we can either use `step` or `next`, but there is a subtle difference between `step` and `next`. If our next line of code is a function call, `next` will consider it as a single instruction and will execute the function all at once. On the other hand, `step` will take us inside the function and go through the code in the function one line at a time. So `step` gives more fine-grained control than `next`.

Also, if you suspect that the function has something to do with the error or bug, you might want to `step` into the function. But if you're sure the function is correct, then you can hit `next` which will run the function and bring you to the statement after the function call.

A.2.2.4 `print` and `display`

To print a value of a variable we can use `print` command.

```
print var   # var is a variable;
print *ptr  # ptr is a pointer.
```

The command `print` will print the value only once. If you want to print the value each time your are in the scope where the variable is defined you can use the command `display`.

A.2.2.5 `watch`

Whereas breakpoints interrupt the program at a particular line or function, *watch points* act on variables. They pause the program whenever a watched variable's value is modified.

```
watch var
```

Note: each time you make any change in the code you need to stop the `gdb` and recompile the program to generate the updated executable (`a.out`). Then run the `gdb` with this new executable file.

ARMv8 Assembly in Action

B

B.1 Program Structure

ARM assembly programs are stored in files with extension `.s`. A complete assembly program has two major segments: `.text` where we mark the beginning of our code, and `.data` which indicates the global data stored in the program.

B.1.1 Our First Assembly Program

The following program is the simplest assembly program, where we just exit the program:

```
1 .text
2 .global _start
3
4 _start:
5     MOV     X0, 0       /* status := 0 */
6     MOV     X8, 93      /* exit is syscall #1 */
7     SVC     0           /* invoke syscall */
```

Line 5 – 7 are the standard procedure to exit any assembly program. What it does is actually to make a system call that can end the execution of the program. We will see other examples in the future, but for now, you only need to remember that always put these three lines at the end of your program, to make sure it can exit successfully.

On line 4 we have a label `_start`, which is where all assembly programs start executing. You can write it anywhere in your code, but it always marks the first instruction in your program. [1] On line 2, we see we declared `_start` as a global label, using `.global`. The first line `.text` marks the text segment, meaning all the lines after it (until other segments) are assembly code.

Both `.global` and `.text` are called **directives**. They are **not** part of the executable—they cannot be translated into machine code and put into the CPU to execute; they are part of the syntax of the assembler, and help the assembler to manage the organization of the program, or with other purposes. All directives start with a dot.

1: Different machines use different labels to mark the start of the program. For example, some machines use `main` or `_main`. For the virtual machine we are using it's `_start`. If you work on different environments, you need to check this information first.

B.1.2 Segments

Assembly programs are managed by segments. In the example above we've seen a `.text` segment. There are also other segments that might be useful in your program.

- `.data`:
  This segment is used to store **global** variables, meaning they can be accessed by any procedures in the program. Local variables used in procedures/functions shouldn't be declared here; instead, they should be directly stored on stack in their frames;

- `.bss`:
  This segment is also used to store global variables, but it's usually used for **uninitialized** data. For example, if you want to reserve a space of 100 bytes in case in your program you need to store some data. In this case, you can declare that empty space of 100 bytes in the `.bss` segment.

Let's see an example below.

```
.text
.global _start

_start:
    MOV     X0, 0       /* status := 0 */
    MOV     X8, 93      /* exit is syscall #1 */
    SVC     0           /* invoke syscall */

.data
    hello_str: .ascii "Hello World!\n\0"
    arr: .dword 13, 24, 1024
```

In this example, we declared a string `hello_str` and an array of double words `arr` in the `.data` segment.

B.1.3 Declaring Data

Again, we emphasize that there's no data type or variable type in assembly language, or in the main memory. Everything is a byte, and different data types are simply different groupings and interpretations when they are brought back to higher-level languages. For the convenience of programmers, however, assemblers typically provide a set of directives that look like declaration of data types. These directives have two purposes: for us, we don't need to care about internal binary representation of data; for assembler, it determines how to translate the data into bytes. For example, if we declare a double word of number zero, the assembler will make the area 64 bits of zeros; if we declare a character of zero, then it'll simply make one byte of zeros.

Table B.1: Directives for different C data types.

C type	Directive
`char`	`.byte`
`int`	`.int`, `.word`
`long`	`.quad`, `.dword`
`float`	`.single`, `.float`
`double`	`.double`
`char[]`	`.string`, `.ascii`

The format to declare data is as follows:

```
label: .type_directive data1, data2, data3, ...
```

Here `label` is used as the address of the first data we declared after it; `.type_directive` is where we declare the type of the data; and the values of the data can be stored after the directive, separated by commas. A brief

summary of different C data types and corresponding assembly directives can be found in Table B.1.

B.1.3.1 Loading Labeled Data

We usually declare data in the `.data` segment as follows:

```
.data
hello:  .quad  1024
```

To use this data, the first step is to load its address to a register, using `ADR` instruction:

```
ADR   X0, hello
```

where we loaded the memory address labeled as `hello` to `X0`. Then we can use `LDUR` instruction to load the data into, say, register `X1`:

```
LDUR  X1, [X0]
```

Note that we cannot use a label itself as base address. For example, this is wrong: `LDUR X1,[hello]`.

> **Trick**
>
> You can use `ADR Xm, .` (a dot as the second operand) to load the current PC, or the address of this `ADR` instruction to register `Xm`.

B.1.3.2 Strings

Strings are declared using `.string` or `.ascii`, with double-quoted characters, such as:

```
str1: .string "Hello"
str2: .ascii  "Hello\0"
```

Notice for `str2` we added a null-terminator at the end. If we declare the string using `.string`, the assembler will automatically add a null-terminator at the end of the string. Alternatively, we can also use `.asciz` where `z` stands for zero (null-terminator):

```
str1: .asciz "Hello"  /* Same to .string */
```

B.1.3.3 Arrays

You probably have already noticed that there's no "array" type in assembly. If we want to declare an array of integers, for example, we simply put all numbers in a row, separated by commas:

```
arr: .quad  12, 34, 56, 78, 90
```

Here we declare an "array" of five long integers. Recall from your data structure class that arrays are simply a list of elements that are logically stored one after another. In fact, when we put a declaration like the one above in assembly, all the elements are exactly stored next to each other, from low address to high.

If we want to index into the array for an element, we need to calculate the byte offset from the base. In the example above, label `arr` points to the address of its first element, so to load an element to a register, we need to first load the base address, and then use the offset to load the element:

```
ADR   X0, arr        /* X0 stores the base address of arr */
LDUR  X1, [X0, 16]   /* The offset of arr[2] is 2*8 = 16 */
```

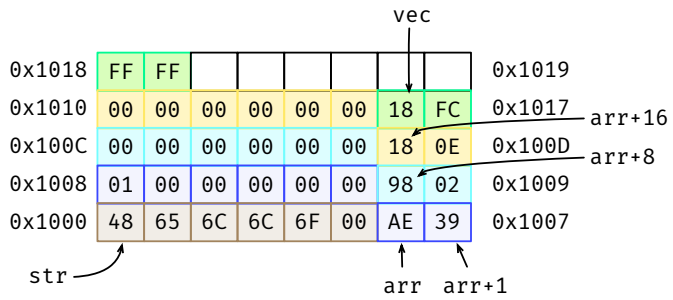
What if the offset is out of the array's boundary? Again, the structure "array" is an abstraction used by us; the internal system of computers does not know anything about "arrays", and therefore there's no such a thing called array's boundary. So if we load an offset of a large number which is outside of the array, the assembler does not issue warnings, and you will probably still be able to run your program without problem. However, as to what data you actually loaded into the register and whether the actual result of your program is correct or not... who knows? Therefore, we as assembly programmers need to carefully manage our own data.

Example B.1

Given the following `.data` segment, draw a memory layout to show how these data are organized. Assume the lowest address of `.data` segment is `0x1000`, and the machine is little-endian.

```
.data
    str: .string "Hello"
    arr: .quad   80302, 01230, 07030
    vec: .int    -1000
```

We show the layout of the memory in the following figure, where each row contains eight bytes. The addresses are increasing from left to right, and from lower rows to higher ones. All the numbers shown are hexadecimal.



The data in `.data` segments are always starting from low address, and store one right next to another. We first have `str` which is a string, so each character in the string will be stored as ASCII code.

Note that in the end we have an additional null-terminator because we used `.string` directive. If we used `.ascii` the null-terminator will not be there.

Next, we have three double words, and they will be stored right after the string. Notice we're using little-endianness, meaning the least significant byte is at the lowest address. After the three numbers, we have an integer, which only takes four bytes.

Previously we mentioned that there's no such a thing as array boundary. As in this example, `arr` can also be referenced as `str+6` (*i.e.*, base address at `str` with six bytes of offset); or `arr+32` is actually pointing to the first byte of `vec`, which is considered by us as "out of boundary".

Assembly doesn't do element check either; once the numbers have been stored there, they are nothing but bytes. For example, if you want to get the first element of the `arr`, but for some reason you added 2 to the base address like this:

```
ADR   X0, arr
ADD   X0, X0, 2
LDUR  X1, [X0]
```

assembly wouldn't give you an error saying, "hey that's the wrong address!". It'll just do whatever you told it to do. In that case, `X1` does not store the actual first element of the array which is `80302`; instead it stores the most significant six bytes of the first element, and the least significant two bytes of the second:

```
01 00 00 00 00 00 98 02
```

so your `X1` will end up being `186899384535875585`. This is because `X0`, after you accidentally added 2, is address `0x1008`, and `LDUR` loads a double word (eight bytes) *starting* from the base address.

B.1.3.4 Alignment

In Example B.1, you probably noticed that when we draw the memory layout we put eight bytes on each row, and each row starts at an address of multiples of eight. The byte pointed by the label `arr`, however, start at `0x1006`, which is non divisible by eight, *i.e.*, not aligned. Having data aligned at a memory boundary in many cases will increase machine performance greatly. In some machine if the data are not aligned it can even generate errors.

To align data to a bounary, we can use `.balign` directive:

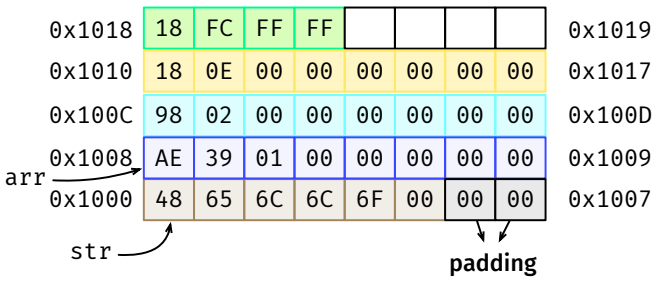


Figure B.1: Using `.balign exp` will align the next data at the address of multiple of `exp`, with zeros as padding.

```
.balign exp
```

which will make the data start at the next address of multiples of `exp`. For example, if `.balign 8` is used, the location of the next data will start at an address of a multiple of 8.

We can modify the `.data` segment in Example B.1 to the following:

```
.data
    str: .string "Hello"
    .balign 8
    arr: .quad   80302, 01230, 07030
    vec: .int    -1000
```

The layout of the memory then is shown in Figure B.1. Notice after the end of `str` we have two bytes of zeros, as paddings, and `arr` starts at `0x1008`, which is a multiple of eight.

B.1.3.5 Repetitive Initialization

In some cases we'd like to initialize an array with the same values. In C, this is how we do it:

```
int arr[10] = {20};
```

where we declare an array of 10 integers, and all of them are 20.

In assembly, we will use a pair of directives: `.rept` and `.endr`., The following example shows us how to use this pair:

```
.data
arr:
   .rept  10
   .int   20
   .endr
```

This also generates 10 integers of 20. Basically, first write the label for the variable; then use `.rept num` to specify how many times you want to repeat the following declarations, and write normal data declarations after it as usual. Lastly, remember to use `.endr` to close the repetition.

Another easier way is to use `.fill` directive:

```
.fill repeat, size, value
```

which will generate `repeat` number of elements; each of the elements takes `size` bytes, and has the value of `value`. For example, again, if we want to initialize an array of 10 integers of 20, here's how to do it:

```
arr:  .fill  10, 4, 20
```

where 4 stands for the size of each element, and so in this case, the size of integers.

B.1.3.6 Reserving Empty Space

Empty spaces are usually declared in `.bss` segment, since it's used for uninitialized data. In fact it's nothing special; if we want to reserve a space of 100 bytes, it's same as declaring 100 bytes of zeros. Two directives can be used:

```
.space  size, fill
.skip   size, fill
```

Here `size` is the **total** size, not an individual element, and `fill` is the data for **every byte** in that space. Say we want to reserve an empty space of 100 bytes in `.bss` segment:

```
.bss
empty_arr:   .skip   100, 0
```

This is to fill every byte of the 100 bytes with zero. It's the same as:

```
      empty_arr: .skip   100
// or empty_arr: .space  100
```

meaning `fill` can be ignored if we just want to fill with zeros. And of course, it's the same as follows:

```
empty_arr:   .fill  100, 1, 0
```

There's no restriction which directive to use in which segment, so you just need to be clear about which parameter stands for what.

Quick Check B.1

We know each byte takes 8 bits, and so if it's unsigned number, the range is from 0 to 256. Now let's do something interesting:

```
.bss
empty_arr:   .skip   100, 1024
```

and see what happens.

B.2 Linking and Executing Assembly Programs

B.2.1 General Workflow

Typically, the three steps to execute an assembly program are:

- **Assemble** source code using an assembler;
- **Link** object files generated by the assembler;
- **Execute** the binary from the linker using an emulator.

Assume we have an assembly source code called `demo.s`. The first step is to generate an object file using the `aarch64` assembler:

```
$ aarch64-linux-gnu-as demo.s -o demo.o
```

If there's no error message, we're all good, and the next step is to link object files to generate an executable, using the linker:

```
$ aarch64-linux-gnu-ld demo.o
```

In case we have multiple object files needed to link to generate an executable, we can just list all of them here:

```
$ aarch64-linux-gnu-ld obj1.o obj2.o obj3.o
```

The output executable name by default is `a.out`, but can be renamed by using `-o <newname>` with the linker. To execute it, use `QEMU` emulator:

```
$ qemu-aarch64 a.out
```

B.2.2 Listing Files

Listing files are very useful to exmaine the format of the object file or even the final executable. It shows you the encoding of each instruction one by one, and how they are arranged and organized in the memory when loaded. Assume we have a source code named `demo.s`. To generate the listing file, we can use the following command:

```
$ aarch64-linux-gnu-as   demo.s  -a=demo.lst
```

where we generate a listing file called `demo.lst`. We usually use `lst` as the file extension for listing files, but it's not required.

The following is an example of a listing file: [2]

```
AARCH64 GAS  demo.s           page 1
   1                .text
   2                .global _start
   3
   4                _start:
   5 0000 000080D2      mov  x0, #0
   6 0004 A80B8052      mov  x8, #93
   7 0008 010000D4      svc  #0
   8
   9                 .data
  10 0000 48656C6C      str: .string "Hello"
  10      6F00
  11 0006 AE390100      arr: .quad   80302, 01230, 07030
  11      00000000
  11      98020000
  11      00000000
  11      180E0000
  12 001e 18FCFFFF      vec: .int    -1000

AARCH64 GAS  demo.s            page 2
DEFINED SYMBOLS
              demo.s:4      .text:0000000000000000 _start
              demo.s:5      .text:0000000000000000 $x
              demo.s:10     .data:0000000000000000 str
              demo.s:11     .data:0000000000000006 arr
              demo.s:12     .data:000000000000001e vec

NO UNDEFINED SYMBOLS
```

On page 1, we see the memory content and the assembly code are listed side by side. For `.text` segments, each instruction's four-byte encoding is shown next to it. For `.data` segment, the binary representations of the data are shown. The four digits in front of them are offsets relative to the beginning of the segment. Also notice directives and labels do not have corresponding memory content in the listing file. This also shows us that they are not part of the program that can be executed by the CPU.

Page 2 summarizes defined and undefined symbols. If, say, we have `BL foo` in our code but we never labeled anything `foo`, the symbol `foo` will appear under "undefined symbols".

One thing about listing file is it will not show all the data declared repetitively. For example, if we declare something like this: `.fill 200, 8, 100`, the listing file will only show the first few bytes.

2: You can compare the memory content of the `.data` segment in this listing file with the example in Section B.1.3.3. It helps you verify the memory layout in that example.

B.2.3 Using External Libraries

We can certainly use some of the functions from the C library in our assembly program, since those C functions eventually need to be compiled into assembly, after all.

B.2.3.1 Examples of Using `printf()`

▶ **Simple Printing**
It is not fun to print values directly in assembly, so one common situation is to print variable values using `printf()`. Since `printf()` is still a procedure, we follow the same rules as branching to procedures. Let's look at the first example below.

```
1  /* print_msg.s */
2  .text
3  .global _start
4  .extern printf
5
6  _start:
7      ADR  X0, hello_str
8      BL   printf
9      MOV  X0, 0   /* Exit */
10     MOV  X8, 93
11     SVC  0
12
13 .data
14 hello_str:
15   .ascii "Hello World!\n\0"
```

In the code listing above, notice we declare `printf` as an external symbol using `.extern` directive. This is to tell the assembler that, "it's ok if you didn't see this symbol defined in this file. It's external, so it'll be in some other files, and will be used during linking." [3]

3: ARM assembler treats all undeclared symbols as external, so it wouldn't be a problem if you didn't use `.extern printf` in your code. However, we do recommend using it to make the code more readable and maintainable.

The call of `printf()` follows the standard procedure call as described in Section 2.6.3. Thus, before we branch to `printf()`, we need to pass the parameters to registers `X0` to `X7`, and even to the stack if more parameters needed. The string we want to print out is declared as `hello_str`, so on line 7, we pass the address of it to `X0` as the first and also only one parameter for `printf()` in this example. Then we just need to branch and link using `BL`.

▶ **Formatted Printing**
Here's another example. Assume we want to print out the value of register `X20`. Written in C, the code should look like this, assuming `X20` stores the value of variable `a`:

```
1 long a;
2 printf("The value in X20 is %ld", a);
```

Here we have two parameters. The first is a formatted string, and the second is a register value. For the first parameter, we need to create a string in the data segment:

```
.data
check_str:  .ascii   "The value in X20 is %d\n\0"
```

For the second parameter, we simply need to copy `X20`'s data to `X1`. The code looks like this:

```
/* print_num.s */
.text
.global _start
.extern printf

_start:
    ADR  X0, check_str
    MOV  X1, X20
    BL   printf
    MOV  X0, 0   /* Exit */
    MOV  X8, 93
    SVC  0

.data
check_str:  .ascii   "The value in X20 is %d\n\0"
```

▶ **More Arguments**

Since `printf()` and all C functions follow the procedure call standards, they can only use the first eight registers for passing parameters. If we want to pass more than eight parameters, we would have to use the stack. In the following example, we want to print five characters and their ASCII codes. With the formatted string, we will have to pass 11 parameters, so `X0`–`X7` apparently is not enough. Therefore, we have to utilize the stack space:

```
/* more_args.s */
  .text
  .global _start
  .extern printf

_start:  ADR  X0,  check_str
         MOV  X1,  49
         MOV  X2,  49
         MOV  X3,  60
         MOV  X4,  60
         MOV  X5,  80
         MOV  X6,  80
         MOV  X7,  100
         MOV  X8,  100
         MOV  X9,  120
         MOV  X10, 120
```

```
17
18          // Assign more space for passing parameters
19          SUB   SP,   SP, 24
20
21          STUR  X8,  [SP, 0]
22          STUR  X9,  [SP, 8]
23          STUR  X10, [SP, 16]
24          BL    printf
25
26          // De-allocate the stack
27          ADD  SP, SP, 24
28
29          MOV  X0, 0
30          MOV  X8, 93
31          SVC  0
32
33    .data
34  check_str:
35    .ascii "ASCII code of char %c is %d, %c is %d, "
36    .ascii "%c is %d, %c is %d, %c is %d.\n\0"
```

On line 19, we allocate a space of 24 bytes on stack to store the three additional parameters. For C functions, regardless of data types, each parameter on stack takes eight bytes. Lastly, remember to de-allocate the frame after returning from `printf()`, as on line 27.

This is yet another example to show you that the machine doesn't care about specific data types and it all depends on our own interpretation of the data. For example, we pass the same number `49` to `X1` and `X2`. Inside the machine, their presentations are exactly the same, but when they are printed out, we see one is the character `1` and the other is the number `49`. This is because we used different specifiers: `%c` and `%d`. Thus, when they are printed out, the `printf()` function takes the bytes needed for different specifiers (1 byte for `%c` and 4 for `%d`), and presents them to the terminal.

B.2.3.2 Pitfalls

One common mistake (more common than you think!) when using C library functions, especially `printf()`, is forgetting about procedure call conventions. From last section it is clear that we need to use `X0`–`X7` and possibly stack space for passing arguments. One thing that we typically forget is that upon returning from the procedure, `X0`–`X7`'s values might be changed by the procedure.

Some students put some values in `X7`, and called `printf()`. After that, they found out that they lost the data stored in `X7`. This is because `X7` is a **caller-saved register**; it is the caller's responsibility to save them in case the called procedure needs to use them. Additionally, procedures need to store return values in `X0` as well. To review this topic, see Section 2.6.2.5.

However, it's not saying all the register values will be changed, but there's no guarantee that they won't be changed, because especially for external procedures, you never know which of them will be used. Therefore, before branching to any procedure, remember to save the data from `X0–X7` somewhere, and restore them after return.

B.2.3.3 Linking C Library

If the assembly code uses the standard C library, we need to use the `-lc` flag to link it:

```
$ aarch64-linux-gnu-ld demo.o -lc
```

where we assume the object file is called `demo.o`.

When executing the program that's been linked with C library, we need to dynamically link it as well: [4]

```
$ qemu-aarch64 -L /usr/aarch64-linux-gnu/ a.out
```

4: On some machines there's no need to dynamically link the library, so if you can run the executable perfectly fine without linking it, you don't have to.

B.3 Debugging using gdb

B.3.1 Installation

The regular `gdb` we used for debugging C programs cannot be used here, because our program can only be executed in `QEMU` emulator, and the architecture is different. We will have to install a `gdb` that can be used for multiple different architectures.

In our virtual environment, we can simply use the following command to install:

```
$ sudo apt-get install gdb-multiarch
```

We will use `gdb` to refer to `gdb-multiarch` from now on.

B.3.2 Start Debugging

We'll use `gdb` and `qemu` together, so we need to open **two terminals** at the same time: one for `gdb` to step through, and the other for `qemu` to provide an emulated environment. If we want to use `gdb` to debug an assembly file, we need to add `-g` flag with the assembler.

Assume the program binary is `a.out`. We first start `QEMU`:

```
$ qemu-aarch64 -L /usr/aarch64-linux-gnu/ -g 1234 a.out
```

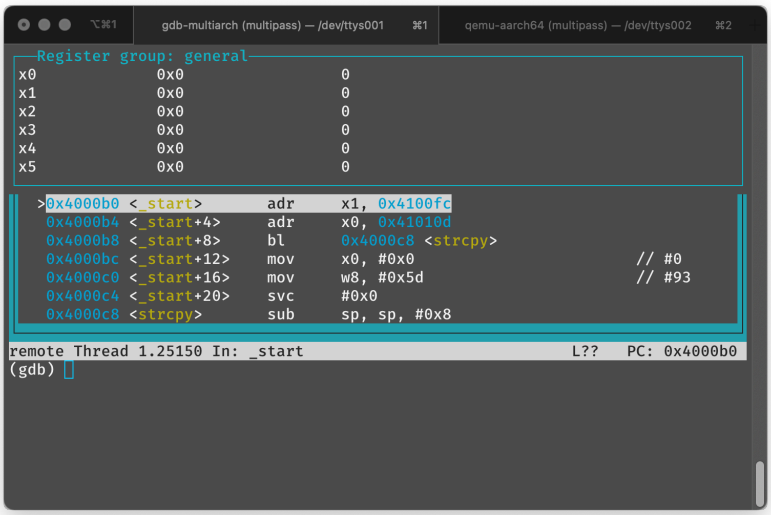


Figure B.2: Interface of running `gdb` for assembly.

where the number `1234` is arbitrary—it's a port for `gdb` to connect. Once this command is in, it'll freeze on the terminal and wait for us to start `gdb`. Now leave it there (do not close it), and we can go to the other terminal and start `gdb`:

```
$ gdb-multiarch --nh -q a.out \
    -ex 'set disassemble-next-line on' \
    -ex 'target remote :1234' \
    -ex 'set solib-search-path
     ↪ /usr/aarch64-linux-gnu-lib/'\
    -ex 'layout regs'
```

Note there's a space between `remote` and `:1234`. The flags `-ex` are the commands we want `gdb` to execute in the beginning. If you don't add these flags when invoking `gdb`, you'll have to type them once you're in the `gdb` environment. The interface you see as in Figure B.2 is called TUI (Text User Interface).

B.3.3 Debugging Commands

`QEMU` doesn't use a typical `run` command to start a program. You can think when you use command `qemu-aarch64` to invoke the program it already started and paused at the first instruction. So simply use command `continue` to start.

B.3.3.1 Breakpoints

When started `gdb`, the program is paused at somewhere in the static library. We can use the following command to set a breakpoint at the beginning of our program:

```
b _start
```

Then we can use `continue` command (or simply `c`) to reach our entry point. Other labels can be set as breakpoint as usual.

Note: if after we set the breakpoint to `_start` and used `continue`, but notice the break point is not exactly at label `_start`, see Troubleshooting section.

B.3.3.2 Steps

Most of the commands are pretty much the same to what we've been using for debugging a C program in `gdb`. As a reminder, when we want to go to a procedure, we would need to use `step` or `s`. If the procedure is from a library, such as `printf()`, it's not a good idea to step into it, so we can use `next` or `n`.

B.3.3.3 Panel Focus

When we enter `gdb` with assembly code and register group laid out, the default focus is the assembly code. Thus, if we press up/down keys on the keyboard, or scroll up/down using a mouse, it only works on the assembly code panel. To change focus to register group to view all registers, we can use command `focus regs`. To change back to assembly code panel, simply use `focus asm`.

B.3.4 Printing Memory

It is quite often that we need to examine the data stored in memory. The syntax is as follows:

```
x/<length><format><unit> address
```

The `length` parameter specifies how much data we want to print from `address`. It can be both positive and negative integers.

The `format` parameter tells `gdb` in what format do we want to see the data. For example, if we pass `x`, it will print the data in hexadecimal. If we pass `d`, it will print in decimal.

The `unit` parameter specifies how to group the data and interpret it.

For the `format` and `unit` parameters, there are many options. Please refer to the `gdb` documentation on `https://sourceware.org/gdb/onlinedocs/gdb/Memory.html#Memory` as well as `https://sourceware.org/gdb/onlinedocs/gdb/Output-Formats.html#Output-Formats`.

The `address` is the starting address in memory. It can be a label, or a hexadecimal address, or a register:

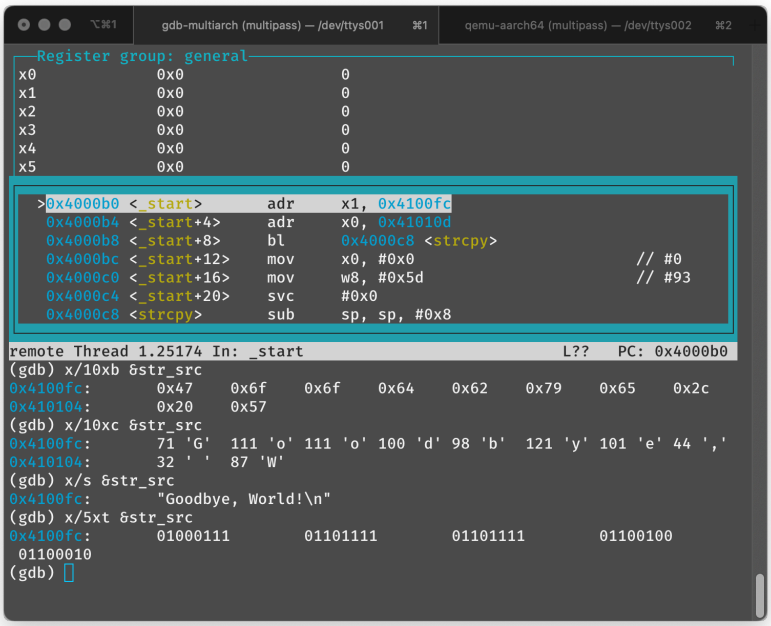


Figure B.3: An example of different memory examining format.

```
# Print 3 bytes in hexadecimal starting from address
 ↪  0x54320:
x/3xb 0x54320

# Print 2 bytes in hexadecimal from the stack pointer:
x/2xb $sp

# Print 2 bytes in decimal from the address stored in x10:
x/2db $x10

# Print 5 bytes in character from the label hello:
x/5cb &hello

# Print the content in a string from label hello until
 ↪  '\0':
x/s   &hello
```

In the example in Figure B.3, we declared a string `str_src` in the `.data` segment. See how different parameters display the same content.

B.3.5 Inspecting Condition Codes

To see condition codes, we can observe `cpsr` field in the register group. CPSR stands for Current Program Status Register. `CPSR` is a 32 bit register, where different flags or conditions take different bits. We only care about the highest four bits: N, Z, C, and V:

31	30	28	28	0 – 27
N	Z	C	V	Other flags

The value displayed in register group panel for `cpsr` is usually in hexadecimal and decimal, so it's not very obvious to see individual bits. We can use the following command to print out the value in binary format:

```
p/t $cpsr
```

where `p` stands for print, and `t` for binary (two's complement), and only look at the most significant 4 bits.

B.3.6 Troubleshooting

B.3.6.1 Breakpoint Not Set Exactly at a Label

You might notice that even if you set breakpoint to `_start`, `gdb` actually set the breakpoint a few bytes later after the label `start`. [5]

To set a breakpoint at the first instruction correctly, you can use the command `info files` to find out the actual address of `_start`:

```
Symbols from "/home/shudong/demo/a.out".
Remote serial target in gdb-specific protocol:
Debugging a target over a serial line.
     While running this, GDB does not access memory from...
Local exec file:
     `/home/shudong/demo/a.out', file type
      ↪ elf64-littleaarch64.
     Entry point: 0x400204
--Type <RET> for more, q to quit, c to continue without
 ↪ paging--
```

Notice in the output above, on line 7, we have the address of entry point `0x400204`, which is location of the label `_start`. Then we can set the breakpoint there:

```
b *0x400204
```

B.3.6.2 No Such File or Directory

If you run the `qemu-aarch64` command to start debugging, but you got an error message like this:

```
/lib/ld-linux-aarch64.so.1: No such file or directory
```

you would need to run the following command to install `aarch64` version of `gcc`:

```
$ sudo apt install gcc-aarch64-linux-gnu
```

5: This typically doesn't happen on M1-based macOS, so try to set a breakpoint at `_start` at first, and if it's not exactly at the label, then proceed to use the actual address to set the breakpoint.

B.4 Floating Point Operations

Real numbers are a different species than integer numbers: they're encoded in a different way; they are stored in a separate group of registers, instead of `X0` – `X31`; they are calculated in a unit called FPU (floating point unit), not ALU (!); and they use a totally different set of instructions.

- **Registers:**
  Real numbers in ARM have 5 types of precision, but we mostly use 2 of them: single (encoded in 32 bits), and double (64 bits). Single precision numbers correspond to `float` type in C, and double precision numbers are, well, `double` type. Therefore, correspondingly, the registers that hold them are `Hn` (half precision), `Sn` (single), and `Dn` (double), where `n` ranging from 0 to 31 is the register number.

- **Usage:**
  Like integer registers, we use `D0` – `D7` to pass parameters to procedures as well as store return values. `D8` – `D15` are preserved across calls, and `D16` – `D31` can be used as temporary registers. The same applies to single and half precision registers.

B.4.1 Basic Instructions

For real numbers, they have a separate set of instructions, but fortunately they are very similar to the ones we know so far.

B.4.1.1 Arithmetic

Typically, when using floating point registers, the instruction has an `F` prefix. For example:

```
1 FADD  S0, S2, S3    // S0  = S2 + S3
2 FADD  D19, D12, D12 // D19 = D12 + D12
```

However, load and store are the same as before.

B.4.1.2 Moving Real Numbers

You can move an immediate real number to an `S` or `D` register using `FMOV`, but there's a restriction on that. Based on ARMv8 manual, only numbers that can be expressed as $\pm\frac{n}{16} \times 2^r$ where $n \in [16, 31]$ and $r \in [-3, 4]$ can be moved. Numbers such as `x.0` (integers), `x.5`, `x.25` are fine.

Therefore, we recommend that you just store all real numbers in the `.data` segment, and load them into registers, and use `FMOV` between registers. Assume we have a number declared as such:

```
.data
pi: .float 3.1415
```

Then to move this number into a register, we should do:

```
ADR   X0, pi
LDUR  S0, [X0]
FMOV  S1, S0
```

B.4.1.3 Converting Precisions

Sometimes we'd like to cast a floating point to a double precision number, or to an integer, or vice versa. We can't just `FMOV` an `S` register to a `D` register. The instruction we need to use is `FCVT`:

```
FCVT    D0, S0   // Upcast, gain precision
FCVT    S1, D1   // Downcast, lose precision
SCVTF   D2, X2   // Convert an integer to a real number
FCVTZS  X3, D3   // Convert a real number to an integer
 ↪ (the fraction part is removed)
```

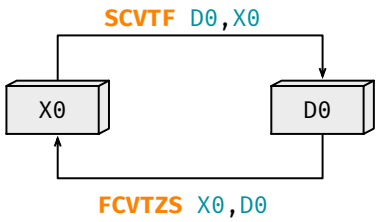


Figure B.4: Converting between double precision real numbers and integers.

See Figure B.4 for an illustration. For other instructions, the best resource out there is ARM64's reference sheet: https://developer.arm.com/documentation/100076/0100/a64-instruction-set-reference/a64-floating-point-instructions.

B.4.2 Printing Using `printf()`

Printing is a little bit tricky. As usual, we need to load a string to `X0`, but the rest of the parameters are passed from `D0` – `D7`.

```
.data
fmt_str: .ascii "%lf %lf\n\0"
number1: .double 3.1415
number2: .double 10
...
ADR   X0, fmt_str   // Load address of the string
ADR   X1, number1   // Load address of number1
LDUR  D0, [X1]      // Load number1 to D0
ADR   X1, number2
LDUR  D1, [X1]
BL    printf
```

When `printf` recognizes `%lf`, it doesn't go to `X1` to fetch the number; instead it goes to `D0`.

If you use `printf` to print both integer values (`int`, `char`, `long int`) and floating point values, move corresponding numbers into their registers in order:

```
.data
fmt_str: .ascii "%d = %lf, %d = %lf\n\0"
...
ADR    X0, fmt_str
LDUR   X1, [...]   // integer #1
LDUR   D0, [...]   // floating point #1
LDUR   X2, [...]   // integer #2
LDUR   D1, [...]   // floating point #2
```

!Caution! In case you declare your number as `.float` like following:

```
.data
fmt_str:   .ascii    "%f\n\0"
fpnum:     .float    3.14
```

you would need to cast `fpnum` from `float` to `double` because `printf` doesn't check `S` registers at all, and automatically uses double precision as output format (even if your format is `%f` instead of `%lf`):

```
ADR     X0,  fmt_str
ADR     X1,  fpnum
LDUR    S0,  [X1]
FCVT    D0,  S0
BL      printf
```

And if you declare your number as `.float`, you cannot load it directly into `D` registers.

Thus, the easiest way to avoid those situations is just declare your number as `.double`, and load it into `D` registers all the time.

B.4.3 Debugging

The floating point registers cannot be viewed in the register panel, so we would have to use `p` to print out:

```
p/f $d0
```

where `/f` is to print out the register value as a floating point.

Alphabetical Index

ARM Assembly Directives & Instructions