# Week 13-14 Module 20, Part 1 – Computer Organization

## Computer Organization Introduction 1.2

Hi. My name is Jerry and I will be your instructor for the Computer Architecture and Organization Section of the CS-Bridge Program. So far, we have covered several paradigms and techniques of computer programming. And these techniques have enabled us to write software code for different purposes. So in this section, we will cover how the CPU interprets this code and executes it. Then we will discuss what is inside the CPU that allows it to execute such code.

More specifically, we'll cover the assembly language and mnemonic machine code, which are human-readable representations of machine-level instructions, we will go over the CPU execution steps, and we will detail the microarchitecture of the CPU.

## A Little Bit of Background 1.3

So, let's cover a little bit of the background on computer organization.  This is just for us to get a flavor of what's inside of a CPU so we can get a better feel for how it executes the code that we write in a language like C or C++. So, at a high-level, there are three main hardware components inside of a CPU: the functional units, the registers, and the memory hierarchy.

The functional units are hardware components that perform logic, arithmetic, and other operations. Examples of functional units we may find are: the Arithmetic Logic Unit , or ALU, which performs integer addition, subtraction and logic operations such as AND, OR, XOR, etc; the floating Point Unit, or FPU, which performs the same operations as the ALU, but for floating point data; the integer multiplication unit, or IMUL, which  performs integer multiplication and division; then there is the floating point multiplication unit, of FMUL, which performs floating point multiplications and divisions.

The General purpose registers or GPRS, are components inside the CPU that hold the data for the functional units. So every data that a functional unit operates on must first be located in a general purpose register. Similarly, once a functional unit finishes an operation, it puts the result back in a general purpose register.

The other important component in the CPU is the memory hierarchy. This hierarchy includes the hard disk, your main memory or RAM, and several other storage components known as caches. The memory hierarchy stores both the software and the data of a given program and different levels of the hierarchy are used for different purposes. For example, the hard disk is used for permanent storage, the RAM and caches are used for temporary storage. We will cover specific aspects of the memory hierarchy towards the end of the computer architecture section

So, let us go through a simple example to illustrate how these three main components work in concert. Let's say we have a simple program, written in C or C++, such as the one we have next to the Memory block. The crux of the code is to perform the addition B+C and to put the result back in memory location of A. Since B and C are integer values 3 and 4 respectively, the ALU functional unit will be used. Remember that each data used by a functional unit must first be available in a general purpose register. So the first operations of the CPU are to move the data for B and C to general purpose registers. So, for the first operation to implement A = B+C is to copy the value of B into a general purpose register. The second operation copies the value of C into another general purpose register. At this point, both values are in registers so the ALU can use them to do the addition. As we see here, in the operation, the ALU performs the addition of 3+4 and puts the result back into a general purpose register. The last operation for this code is to copy the value of the result register back into the memory location for A. So in total, we have four operation to perform A = B+C. The first two operations copied the values of B and C into the registers, because the functional units only operate on data that are in those general purpose registers. The next operation performs the addition and puts the result into a general purpose register, and the last operation copies the value of the result from its register to memory. Later in this section, we will cover this example in more intricate details

## A Little Bit of Background 1.4

So, let's quickly go over two types of CPU architectures in the market. One is called Reduced Instruction Set Computing or RISC, and the other is called Complex Instruction Set Computing, or CISC. The decision of which of those architectures to use impacts how the machine-level instructions of the CPU work, how the hardware and data path of the CPU are implemented, and how the compiler for that CPU works

## A Little Bit of Background 1.5

Let us first discuss the RISC architecture. The basic idea of RISC is to have simple instructions that perform simple operations. Let us recall the A = B+C function from before. In a RISC processor, each of the four operations needed to implement this function can be implemented as a machine-level instruction. So that is four RISC machine-level instructions for one high-level C or C++ line of code. So there would be an instruction to load B to register, and instruction load C to register,  an instruction to add registers for B and C and put the result back in a register, and an instruction to store the result back to memory allocated for A.

The benefit of having instructions that perform simple operations is that they make it easy to implement the CPU at the hardware level. So the CPU hardware is less complex and incurs less power and space in your computer. The main downside is that it is more difficult to translate the high-level code into machine-level instructions. This makes it more difficult for the compilers. The most popular RISC architecture used in the market is ARM, which is commonly found in smartphones, tablets, and low-battery platforms

## A Little Bit of Background 1.6

CISC architecture does the reverse of RISC. That is, it has very complex machine level instructions, where an instruction can perform multiple high-level operations. For example, an instruction can do both a memory access operation and an addition operation.

The main benefit of this is that it makes it easier to translate high-level code, such as C or C++ into machine instructions, simplifying the compiler. The downside is that the hardware of the CPU is more complex because of the multiple operations that an instruction can perform. The most popular CISC architecture is x86 from Intel.

## What We Have Covered 1.7

So we have gone over a high-level view of the most important components of a generic CPU, discussed general purpose registers and the functionality and we have differentiated the RISC and CISC computer architecture. In this class, we will use a RISC architecture known as MIPS to cover the different topics.

## Assembly Language (Pt.1) 2.1

In this module, we will introduce some basic concepts of assembly programming and then correlate it to machine-level instruction using a mnemonic format. More specifically, we will introduce some basic assembly instructions in the MIPS architecture. We will then go in detail on how to map MIPS assembly into MIPS mnemonic format, which is the format we will be focusing on throughout the class, and we will cover some registers available in a typical MIPS architecture

## MIPS Instruction Set Architecture 2.2

So in this class, we will use the MIPS processor architecture as reference. MIPS stands for Microprocessor Without Interlocked Pipeline Stages. MIPS uses RISC architecture, is open-source and is commonly found in gaming systems such as Playstation 1, 2, Playstation portable, and the like.  In this class, we will use MIPS64 architecture. Each MIPS64 instruction is 32 bits long. The default word size is 32 bits. However, the 64 and MIPS64 indicates that the processor can do operations on data as large as 64 bits, or double word.

## MIPS Assembly: Introductory Example 2.4

So, let's go through a detailed example of converting a high-level code in C or C++ to MIPS assembly and ultimately to MIPS mnemonic format. Consider the high-level code we have here as an example. Note that the values for A, B, and C, are initialized in hexadecimal format

So when we successfully compile this high-level code using our MIPS compiler, we get the software binary that is compatible only to a MIPS processor. This binary is the executable file of the code. And if we try to open this binary, what we see is a bunch of ones and zeros representing the bits for the machine-level instructions that implement this high-level code

This binary itself is generally broken down into two segments: a code segment that contains the machine level instructions that implement the functionality of the high-level code and a data segment that contains the declared static variables in the high-level code. Each segment has a start address that we call the base address. Note that the addresses are given in hexadecimal. In our example, the code segment base address is F00 and the base address of the data segment is A000. Note that in the data segment, each static variable is initialized to 64 bits. This is the default configuration of the MIPS64 processor.

So ideally, what we would like to do is to be able to understand how the high-level code A = B+C maps to machine code However, this is too complicated because working with the bitwise representations of machine-level instructions is tedious and error-prone

## MIPS Assembly: Introductory Example 2.5

Instead, we start by working with assembly language. The assembly language is a symbolic representation of the machine code. In this respect, symbolic means it simplifies some of the complexities of machine code by using labels and symbols. Nevertheless, given a machine-level code, there is a one-to-one mapping between a machine instruction and an assembly instruction

## MIPS Assembly: Introduction Example 2.6

Let's illustrate this with the A = B+C example. Recall that MIPS64 is a RISC architecture, so there must be four machine-level instructions to perform A = B+C: two instruction to load B and C to registers, one to

add the registers, and one to store the result back to memory location of A. Since there is a one-to-one mapping between machine code and assembly, we can convert this high-level code into four assembly level instructions. The first instruction copies the value of B from the data segment to a register; let us assume the assembly labels the register using the symbol t1. The second instruction copies the value of C from the data segment to another register, say register t2. At this point, the value of B and C are in t1 and t2 respectively. The third instruction adds the the registers t1 and t2 and puts the result into register t3. The last instruction copies the value of t3 into the data location that is allocated for A.
Here we see the more formal assembly language code for this high-level operation. The first two assembly-level instructions are load double word, or LD. In this respect the double word indicates to the processor that the data to be loaded from memory to register is of size double word, or 64 bits. The third instruction is a double word add, or D--DD, where the ALU adds the two registers t1 and t2 and puts the results in t3. The final instruction is store doubleword, or SD, where the content of t3 is copied into memory address of A. Note the syntax of the assembly-level instructions. For load doubleword, we see the destination register is put first, followed by the label of the memory address. For the doubleword add, the destination register is put first, followed by the source register operands. For the Store Doubleword, the source register is put first, followed by the label of the destination memory location.

So, since assembly language has a one to one correlation to machine code, we can map each assembly instruction of the A = B+C operation to a machine-level instruction.

## MIPS Assembly: Introductory Example 2.7

However, assembly code doesn't fully represent machine-level instructions. This is due to symbolic notations of the registers and the memory addresses. The processor doesn't understand a memory address labeled as B or a register labeled as t1.

Let us consider the first Load doubleword instruction we had earlier and let's compare it to the real machine instruction. Using this example, we will see that the assembly notation doesn't have some of the fields of machine-level code.

So in MIPS64 architecture, the most significant six bits of a machine-level instruction represent its opcode. The opcode is a simple mnemonic notation to encode the operation of the of the instruction. In our example, the opcode 110111 means this is a load double word instruction. The assembly instruction also has a load doubleword opcode.

The next 5 bits of the machine-level instruction represent the base field. This is the number of a general purpose register number that holds the base address of the data segment. In our example, this register is register number 8, or R8. Note that there is no such base register field in the assembly instruction. Instead, it simply uses the label name of the data as it was declared in the high-level code. The CPU cannot understand such high-level label.

The next 5 bits of the machine-level instruction represent the Rt register field. This field indicates the destination register where the double word will loaded. In the machine-level instruction, this Rt register is register number 9 or R9 and is labeled as t1 in the assembly code. This is still similar to the assembly level code, though the labeling is different

The last 16 bits of the machine code is the offset field. It indicates where within the data segment the requested value is located. In our example, the offset is 0 because B is the first value in the data segment. Note that the assembly-level instruction does not have such field.

## From Assembly to Mnemonic Format 2.9

Although assembly-language provides a one-to-one correlation with machine code, it still has some limitations with fully illustrating how the CPU interprets the machine-level instructions. And since this is a computer architecture and organization course, we want to have as good of a feel as possible of the CPU inner-workings.

With that in mind, we use MIPS mnemonic format to represent software code. We can think of mnemonic format as a readable version of machine-level code. It interprets all of the fields of machine-level code to better illustrate the inner-workings of the CPU.

When writing in MIPS mnemonic notation, we generally use hexadecimal format, except for register names. For some cases, the hexadecimal notation of number may take too much space. In such cases, we use the decimal notation of that number and explicitly say it is the decimal notation. For example FFFF in hexadecimal is equal to -1 in decimal, so we put -1 in parentheses and add the 10 subscript to indicate this a decimal notation. This is just to make it easier to write MIPS in mnemonic format.

## Introductory Example 2.10

So here we see the MIPS mnemonic version of our working assembly example.  We notice some clear differences from the offset. The first thing we notice is that each instruction has an associated address. This is the address of the instruction in the code segment. Note that these addresses are given in hexadecimal format.

Recall that since each instruction is 32 bits, or four bytes, it is easy to calculate the address of each instruction once we know the base address of the code segment. The base address itself is the address of the first instruction in the segment. We also notice that the registers are given using numerical notations, as is done in machine code.  There are 32 general-purpose registers in MIPS64, so they are labeled R0 to R31.

The last thing we notice is that the mnemonic notation has the same fields as the machine code. Using the first Load doubleword example, we see that the mnemonic notation has the base and offset fields, just like the machine code discussed earlier

## Introductory Example 2.11

Here we put the full version of the code segment in MIPS mnemonic notation. The semicolon indicates the beginning of a comment. So anything following the semicolon is a comment. This is similar to the forward slashes in C or C++.

Regarding the data segment, we know that each double word data is 64 bits, or 8 bytes, because we are using MIPS64. So once we know the base address of the data segment, we can find the addresses of all data in the segment. Note that this is the same approach that the CPU uses to determine the address of data that it accesses. That is, the CPU uses the base address of the data segment and the offset of the target data within the segment. So putting it all together, we get the following notation for the code and data segments in MIPS mnemonic. Note that all addresses are in hexadecimal.

## What We Have Covered 2.12

We have introduced MIPS assembly language and covered some fundamental instructions. We have also seen that, although assembly has a one-to-one correlation to machine level code, it doesn't fully illustrate how the CPU executes the code. We have then covered MIPS mnemonic notation, which gives a good middle ground between assembly and machine-level code. Next, we will use a tabular approach to illustrate how the CPU updates registers and memory when it executes instructions

## Assembly Language (Pt.2) 3.1

We'll now pick up from the MIPS mnemonic notation and start discussing some aspects of CPU execution. We will use a table to show how the CPU updates registers and memory when executing each instruction. We will then discuss the three instructions we have covered so far and detail their fields and how they work. Finally, we will review the 32 MIPS general purpose registers and what their roles are

## MIPS Assembly: Introductory Example 3.2

We will now show how to execute this MIPS code in table representative of the components of the CPU. So the table has the following columns: an instruction column that lists the instructions in MIPS mnemonic format; a PC column, where PC stands for program counter. The Program Counter is a special register in the CPU that holds the address of the next instruction to execute. Then, there is a column for each register used in the MIPS mnemonic code. Since our example uses registers R8, R9, and R10, there is a column for each. There is also a column for each data in the data segment. Since there are 3 pieces of data in the data segment, there are three columns and we use their addresses to label each. The last column is a memory access column to indicate what kinds of memory accesses the associated instruction makes. Since we are using the MIPS mnemonic format for the table, the content of the table will be in hexadecimal, unless stated otherwise using the subscript notation.

The first column of the table is ALWAYS the Initial column. This column illustrates the state of the CPU before execution of our code. At the initial state, the data segment has the original data of the variables in the code. So we write those values in their respective columns in the table.

In addition, during initialization the CPU writes the address of first instruction in the Program Counter and writes the base address of the data segment in the R8 register. Remember that the Program Counter stores the address of the next instruction to execute. So since the next instruction to execute is the first one, it make sense that PC has that address before any instruction is executed. Since we don't initialize the values of the other registers, we set them as question marks.

For each instruction, the first thing the CPU does is it reads the instruction from the code segment; that is known as an instruction fetch or instruction read.  We concatenate it as INS read in the memory access column. The CPU uses PC to get the address of the instruction to fetch

Since the Program Counter points to the address of the next instruction, once an instruction is fetched, the Program Counter has to be updated to point to the next instruction. Generally, this update is done by simply incrementing the Program Counter by 4, because instructions are 4 bytes long. Later in this course, we will when Program Counter is not always incremented by 4 due to branch and jump instructions.

We then analyze the instruction to see what it does and update the columns of the table accordingly. In our example, the instruction is a load double instruction, where the base address of the data segment is in register R8, the offset is 0, and the destination register is R9. So, the CPU first calculates the address of the data in memory. This is done by adding the content of the base register to the offset. Once the address is obtained, the CPU reads the memory at that address to get the data. Since this is a memory access, we update the memory access column to indicate a data read. Once the value data is obtained from memory, the CPU writes it in the destination register.

For all registers and memory locations that are not modified by this instruction, we put an NS label to indicate Not Stored. In other words, this instruction does not store anything in those registers and data locations and their values remain the same as before

## MIPS Assembly: Introductory Example 3.3

So at the end of this instruction, we see the value of R9 has been updated as desired by the Load double word and the program counter points to the next instruction. The next instruction is also a load double word and we see that the memory access column indicates an instruction read to fetch the instructions from the code segment, and a data read, to read the double word data from the data segment. In addition, we see that the Program Counter is incremented by four to point to the next instruction to execute and we see that the destination register R10 of the load doubleword is updated with the appropriate value. Everything else remains unchanged.

The next instruction is a Doubleword add. This instruction takes the content of registers R9 and R10, sends them to the ALU for addition, and puts the result in the destination register R11. The instruction only does an instruction read because it doesn’t access the data segment. Its operands are only registers. We see that the Program Counter is incremented to point to the next instruction and R11 has the result of the addition in hexadecimal.

The last instruction is a store doubleword. It works in a similar fashion to load doubleword, except that instead of loading memory to register, it stores from register to memory. Store doubleword uses its base register and offset to calculate the target address of the memory, then writes the content of the register operands to the memory location indicated by the calculated address. As a Store doubleword, the CPU does an instruction read and data write for this instruction. We also see that the PC is incremented by 4, even if there is no such instruction at address F10 in our code segment. This is because the CPU does this increment automatically.

At the end, this is content of the table. It illustrates the different registers and memory locations that are relevant to the code and how they are impacted by each instruction in said code.

## MIPS Assembly: Introductory Example 3.4

Let us now discuss the syntax and semantics of the Load doubleword, doubleword add, and store doubleword instructions that we have already covered. Here we see the syntax and semantics of the Load doubleword. Rt is the destination register, Rs holds the base address of the data segment and the offset holds the position of the data relative to that base address. The offset is always at most 16 bits. To calculate the address of the target data, the CPU adds the value of the base address in Rs to the offset. But Rs is a 64-bit register, and the offset is 16 bits. The addition requires both values to  be the same size.  To change the size of the 16-bit offset to 64 bits, we perform what is called a sign extension.

The basic idea of a sign extension is to increase the number of bits used to represent a value, while keeping the sign. So the naïve approach to extend a 16-bit value to 64 bits would to append 48 zeros to the left of the original 16-bit value.  However, this approach doesn't work if the number is negative.  This is because if we extend a negative number with zero bits, then it becomes positive because the most significant bit is now 0 instead of 1. So sign extension works by first looking at the most significant bit of a binary, and using the value of that bit to extend it to the desired bitsize.  The three examples here show how to sign extend different 16-bit values to 64 bits

## MIPS Assembly: Introductory Example 3.5

The double word add instruction adds two registers Rs and Rt and puts their result into Rd register. The ALU may need to perform some sign extension to ensure that the content of both Rs and Rt are 64 bits

The store doubleword instruction has a similar syntax as load double word but the semantics are obviously different. Store doubleword copies the content of register Rt to the memory location indicated by the address calculated. Similar to load doubleword, the calculation of the address requires the offset field to be sign extended to 64 bits.

## MIPS General Purpose Registers 3.7

Let us now go over the list of general purpose registers in the MIPS CPU. Remember there are 32 of them, labeled R0-R31. R0 always contain the value 0 and it's read-only. That is, no code can write to it. It is only used whenever we want to use the value 0. R1 is reserved for the assembler. R2 and R3 are used to return results of function calls. If you have a function that returns an integer, then it will be stored in either R2 or R3 after the function completes. One interesting question is if we have a function that returns something like a struc of 20 integers,  how does MIPS return the 20 values when it only has 2 registers. I will leave it to you guys to find out. R4 to R7 are used to pass parameters of a function call. Similarly, I will let you guys find out how to call a function that passes more than four parameters. R8 to R15 and R24 to R25 are used to store temporary values that are declared in a function. So when then function completes, the values of these registers are not saved. We will primarily use these registers in our MIPS mnemonic codes. R16 to R23 are also used to store values in functions, but they are stored and saved when the function completes. So they can be used across multiple functions. R26 and R27 are reserved for the O.S. R28 is the global pointer and it points to the static data segment of the code. R29 and R30 point to the stack and frame pointers respectively. The stack and frames are structures in the CPU that store local variables of a given function. R31 holds the address to return from a function call. For our course, we will primarily use R0, R8 to R15, R24, R25, and R31. Note that we do not include the

Program Counter in this list because it is not a general-purpose register. It is a special register and cannot be accessed by any user-level code.

## Another Example 3.8

As an exercise, I am encouraging you guys to try the following exercise. First, write the MIPS Mnemonic code that performs the high-level code above. This code does A = B or C. Note that this is a logic or operation. Once you have written the mnemonic code, show its execution in the table as we did before for A = B+C example. The code and data segment addresses are specified, as well as the original values of A, B, and C.

## Conclusion 3.9

I have added a document on the course site that summarize the 32 MIPS general purpose register that we have discussed earlier. You guys don't have to memorize their purposes, this is just as a reference. I also added the official documentation of the MIPS64 instruction set architecture. If you guys have any questions regarding anything we discussed, don't hesitate to send me an e-mail. Thanks.