Week 13-14 Module 20, Part 2 – Computer Architecture

Outline 1.2

Hi everyone. Today, we will cover the second half of the computer architecture section of this course. We will focus on branch and jump instructions, which are fundamental in all instruction set architectures. We will discuss the purposes of these instructions, how they are mapped to high-level C and C++ codes that we write every day, and we'll go over two examples of their use cases

Instructions Covered So Far 1.3

So far, we have covered basic instructions such as Load and store instructions to get data from and to memory, as well as some arithmetic and logic instructions. The arithmetic and logic instructions we have covered so far require that all operands to be first put in registers. However, there are cases when we write code and we use hardcoded values, instead of variables, to perform arithmetic and logic operations.

As the high-level code example highlights, the second variable is hardcoded as 17. MIPS64 provides Immediate Arithmetic and Logic Instructions for such high-level code. The basic idea of an immediate ALU instruction is to store the value of the hardcoded operand as part of the 32-bit instruction. Several immediate ALU instructions are provided including doubleword ADD I, for addition, doubleword OR I for OR operation, and doubleword AND I for AND operation. The I at the end of each of these instructions stands for immediate

Each immediate ALU instruction has a 16-bit immediate value to store the hardcoded operand. This 16-bit immediate is labeled, do I M M, for Do Immediate in the semantics. For an immediate arithmetic instruction such as doubleword ADD I, this do immediate value must first be sign-extended before the arithmetic operation is performed. However, for an immediate logic instruction such as D OR I or D AND I, the do immediate value does not need to be extended. It simply needs to be padded with zeroes to reach 64 bits before the logic operation is performed.

Knowledge Check 1.4

Answer A

That is correct. First, you took negative 4, convert it into hexadecimal which is FFFC. Then we took the original value of R8 add it to our hexadecimal value of negative four, and the answer is correct.

Answer B

Remember, here the immediate value is negative four. So you must first convert the negative four value in its representative hexadecimal format, sign-extend it, then do your addition.

Answer C

Here, you have to keep the size of the operands consistent. Remember, this is a DADDI instruction, so the ALU is going to use double word, or 64-bit values. Therefore, you must sign-extended both the register R8 and the immediate value to 64 bits before you do the addition.

Answer D

Pay close attention to the semantics of the instruction. The register at the Rt position is the destination and the register at the Rs position is the source. For our case, the source is R8.

Jump Instructions 1.5

We will now discuss Jump instructions. So far, the instructions we have covered only allow us to write sequential code. But what happens if we have function calls, conditional statements, and loops in our code. As we know, code that we write generally involves these situations.

This is where jump instructions come into play. These instructions allow the CPU to skip chunks of sequential code in the binary and to jump to locations specific… to jump to locations of specific instructions in that binary. We call these locations target instructions. There are two types of jump instructions: unconditional jumps, where we always jump to the target instructions, and conditional jumps, where we branch to the target instructions based on a given condition.

From now on, we will refer to unconditional jumps simply as jumps and conditional jump as branches. Jump instructions are generally used for calling functions and returning from them, whereas branch instructions are used for if/else statements and conditional loops

Another important factor in jump and branch instructions is how far can the instructions jump and how to calculate the target address of that instruction we are jumping or branching to. There are generally two methods: one is called relative branch or jump. In this method, the target address is a function of the program counter of the branch or jump instruction. The other method is called indirect branch, where the target address is independent of the program counter.

Relative branch and jump instructions restrict how far the target addresses can be, usually it is bounded within a few kilobytes of the program counter. On the other hand, indirect branch and jump instructions obtain their target addresses from registers or a memory and can, in theory, take any value.

Knowledge Check 1.6

Answer A

Remember, by default, the CPU goes to PC+4 for non-branch and non-jump instructions. Therefore, there is no need to use a jump instruction to do PC+4. Jumps and branch are thus NEVER used to go to the next instruction in the code sequence.

Answer B

When the application finishes its job, it should, in fact it must, jump to the function that calls it. This is usually an operating system function, or another application. So it wouldn't jump to its own code if it has finished executing.

Answer C

That is correct. If the unconditional jump instruction jumped to itself, then the target address will be again the unconditional jump instruction. Therefore, the unconditional jump instruction will continually jump to itself over and over again resulting in an infinite loop.

Answer D

Remember, jump instructions are unconditional. So there is no condition to check. The CPU will ALWAYS go to the calculated target address.

MIPS64 Unconditional Jump Instructions 1.7

Let us now detail the MIPS64 jump instructions. There are three such instructions: Jump, Jump and Link, and Jump register. Jump is commonly used when performing unconditional jumps within a function or procedure. This can be used for example for goto statements. Jump and Link is used for function calls and has two jobs: it first saves the return address in the R31 register, and then updates the program counter to the address of first instruction of the function. Note that the return address in this case is the address of the instruction we must return to when the function call has completed.

Jump register is used to return from a function call. Jump and Jump and link both use relative jump, whereas Jump register uses indirect jump and holds the address of the target instruction in a register.

MIPS Unconditional Jump Target Address 1.8

Let us now discuss how to calculate the target addresses of the different jump instructions. As mentioned before, Jump register uses indirect jump, where it gets the target address from a register. Therefore, there is no calculation done, as the address of the target instruction is simply copied from the register RS to the program counter

On the other hand, both Jump and Jump and Link use the same function to calculate the target address. We label this function func1 and it uses the program counter of the jump instruction and another 26-bit parameter that we call address. Note that this address parameter is not the target address we are looking for.

Let's take a close look at the formula for func1. The first step is to increment the program counter by 4. In this instance, the program counter is the address of the jump or the jump and link instruction. The next step is to take the most significant 4 bits of the result from step 1. This assumes that the program counter is 32 bits. If we encounter a case where the program counter is less than 32 bits, then we have to pad it with zeroes to make it 32. The third step is to take the address parameter… to shift the address parameter left by 2, which has the same effect as

multiplying it by 4. This turns the 26-bit parameter to a 28-bit parameter. The last step is to concatenate the most significant 4 bits from step 2 with the 28 bits from step 3. The result is the 32-bit address of the target instruction.

Let us take a look at an example of how calculate a target address for jump and link instructions. Here we have a jump, where the program counter is 4DC and the address parameter is 13B. Note that all numbers are in hexadecimal. We expand the program counter to 32 bits because the formula for func2 expects 32-bit program counters. The first and second steps are straightforward; this is because even after we do PC+4, the most significant 4 bits of the program counter are still 0.

For step 3, we multiply the address parameter 13B by 4. Remember that this is a hexadecimal multiplication, so the correct answer should be 4EC. Then we expand the multiplication result to make it 28 bits. This is done by padding the result with zeroes. The last step concatenates the results from steps two and 3 respectively, giving us the target address highlighted.

Knowledge Check 1.9

Answer A

Pay close attention to the formula for func1. Note that the PC+4 addition is a hexadecimal addition. Also, the PC is assumed to be 32 bits

Answer B

Pay close attention to the formula for func1. Note that the PC+4 addition is a hexadecimal addition. Also, the PC is assumed to be 32 bits

Answer C

Remember, you must take the most significant four bits of PC+4. The formula assumes that the program counter is 32-bits. So if it's not, you must first pad it with zeroes to make it 32 bits. Then take the most significant four bits [31:28] of the result

Answer D

That is correct. We first took the address parameter, which is 13 bit, shift it left by 2 which has the same effect as multiplying it by 4 given us four EC. Since the most significant four bits of the program counter is zero, then the final answer itself is 4EC

MIPS64 Conditional Branch Instructions 1.10

Let's now go over branch instructions in MIPS64. The two most fundamental branch instruction in MIPS64 are Branch if equal, or BEQ, and Branch if not equal, BNE. Both branch instructions use relative target calculation and have the same Function, labeled func2, to calculate the address of the target instruction. Both instructions compare the values of two registers to check their condition. Branch if equal checks if the values of the two registers are the same. If so, the CPU branches to the calculated target address. Otherwise, the CPU goes to the next instruction in the sequential code which is PC+4. Branch if not equal checks if the values are different. If that is the case, the CPU branches to the calculated target address

Several other branch instructions derived from branch if equal and branch if not equal. Such instructions are Branch if equal to zero, branch if not equal to zero, branch if less than zero, branch if less than or equal to zero, any many more. We focus primarily on branch if equal, branch if not equal, and branch if equal to zero. There is one terminology that is commonly used when talking about conditional branches. It is the “branch is taken/not taken” term. When one says a branch is taken, it means that the condition that the branch checks is true and the CPU branches to the target address calculated. Branch not taken means the opposite.

Knowledge Check 1.11

Answer A

When we say a branch is taken, what we mean is that the comparison of the branch condition is true. We don’t necessarily mean that the registers are equal. So if the branch checks that two registers are not equal, such as branch if not equal does, and the registers are indeed not equal, then the comparison of the branch condition is true and the branch is taken

Answer B

That is correct. The branch not equal instruction checks if the two registers are not equal; in our case r8 is 3 and R0 is always 0. So the two registers are indeed not equal, so the branch condition is true and the branch is taken.

Answer C

R8 is equal to 3 in our example. R0 is ALWAYS equal to zero; therefore the two registers are not equal

Answer D

R8 is equal to 3 in our example. R0 is ALWAYS equal to zero; therefore the two registers are not equal

MIPS64 Conditional Branch Target Address 1.12

Let's go over the func2 formula that calculates the target addresses for branch instructions. Since branches use relative address calculation, the formula uses the program counter of the branch instruction. In addition, the formula uses a 16-bit immediate parameter that we label as do immediate.

Here is a little bit more detail on the formula for func2. It first takes the program counter and increments it by 4. Recall that in this situation, the value of the program counter is the address of the branch instruction. It then takes the 16-bit immediate value and sign-extend it to 32 bits. In the third step, the formula takes the sign extended immediate value and shift it left by two. This has the same effect as multiplying it by 4. The final step adds the results from step 1 and 3 to get a 32-bit target address.

Here is an example to illustrate how func2 works. We have a branch if equal instruction with the program counter and the do immediate parameter given. The first step adds 4 to the program counter and the second step sign extends the do immediate value. Since the Do Immediate is a

positive value, the sign extension simply pads zeroes to make it 32 bits. The third step takes the sign-extended Do Immediate and multiplies it by four, and the last step adds the results of step 1 and step 3 to get the target address highlighted

Knowledge Check 1.13

Answer A

We first check if the branch condition is true. R8 is 3 and R0 is always 0. Since the branch condition checks if the registers are not equal, it is indeed true. Therefore, the CPU goes to the target address and doesn't go to PC+4. Use the formula in func2 to calculate the target address.

Answer B

First, we compare the branch condition. Is R8 not equal to zero? Yes. So the branch is taken and the CPU branches to the target address. Now use the formula in func2 to see if the target address is the end of the application

Answer C

That is correct. We first check if the branch condition is true; the registers are not equal, so the branch condition is true and the branch is taken. To calculate the target address, we take the immediate value, sign extend it which gives us all F's because it is a negative one in decimal. Multiply it by four which should give us FFFFC. And we add it to the value of the program counter which should give us PC+4, which should give us back the original value of the branch if equal program counter. Hence, the branch instruction jumps back to itself resulting in an infinite loop.

Answer D

The branch condition can be determined from the information available. We know that R8 is 3 and R0 is always zero. The branch checks if R8 is not equal to R0. In our case, the branch condition is true.

MIPS64 SLT Instruction: Set Less Than 1.14

One instruction that is strongly correlated with branch instructions is SLT or Set less than instruction. SLT uses three registers, Rd, Rs and Rt. It checks if Rs is less than Rt. If so, it sets Rd to 1. Otherwise, it sets Rd to 0

Several branch instructions in MIPS64 are formed by combining SLT with either branch if equal or branch if not equal. For example, Branch if less than zero can be formed by SLT and branch if equal as the example here shows. A similar method can be used for Branch if greater than zero instruction.

MIPS64 Branch/Jump Instruction Uses 1.15

Let us now see some use cases of branch and jump instructions. Consider this high-level C code. This is a function that calculates the absolute value of a number. That is, if the number is negative, it turns it to a positive; otherwise, it keeps it as is. The value us returned from the functional call. As we see from the high-level code, there is an if condition to check if the

number is negative. This if condition in high-level code will be translated to a conditional branch instruction. We also notice there is a return instruction to end the execution of the absolute value function. This return instruction will be translated to a jump register, as it is the jump register instruction that is used to return from function calls.

Here are several approaches to translate the if X is less than 0 condition into machine code. Let us assume that X is in register R4. The simplest approach is to use the branch if less than zero instruction. In that case, if X is negative, the CPU would skip the sequential instructions and branch to the part of the code that perform X = minus X in order to get the positive value. Another way to is to use the branch if greater or equal to zero condition. In that case, X is positive, we would skip the sequential instructions and branch to the part of the code that does return X. Otherwise, we would subtract.

Another way is to combine SLT with branch if equal instruction. The SLT instruction checks if X is less than zero. If it is, it set R8 to 1. Otherwise, R8 is set to zero. The branch instruction then checks R8 to zero. If R8 is equal to zero, then x is positive, the branch is taken, and the CPU branches to the return instruction. We will use this approach to further develop the code for the absolute function.

MIPS64 Branch/Jump Instruction Uses 1.16

So, now we have the code that handles the if condition. How do we write the rest of the code to complete the function? First, what instruction should be right after the branch instruction? In the sequence of the code, the instruction following the branch must handle the case where the branch condition is false and the branch is not taken.

So in our example, the branch instruction condition checks if X is positive. Therefore, the next instruction in the sequence handles the case where X is negative. For our absolute function, when X is negative, we need to negate it. For that, we can simply use a doubleword sub or DSUB instruction to negate R4 and turn it into a positive value.

To find out what the instruction after the subtraction should be, we again look at the high-level code of the absolute value function. We observe that once the subtraction is performed, the next operation is to return from the function call. And we already know that if the branch condition is true and X is already positive, we would simply return from the function call. In this respect, there is nothing left to do in this code but to return from the call. For this return instruction, we use the jump register. This jump register instruction is also where we would do the branching to in case our branch if equal condition is true.

MIPS64 Branch/Jump Instruction Uses 1.17

The code is still not complete. More specifically, we need to replace the Do immediate parameter with its correct value in the branch if equal instruction. And we need to set the correct value of Rs in the jump register instruction.

To calculate Do immediate, we can simply use the inverse of the func2 formula. This is the formula that we use to calculate the target address of branch instructions. Based on the inverse formula of func2, we use the values of the target address and the address of the branch instruction to calculate the do immediate. Our result shows that Do immediate is 1.

MIPS64 Branch/Jump Instruction Uses 1.18

To set the correct value of the Rs in the jump register instruction, we need to first understand the interaction between the jump and link instruction the and jump register instruction. Consider this use case of the absolute value function. We have a header file Lib dot H that holds the declaration of the absolute value function and a c file Lib dot C that holds its implementation. In the main file, we call the absolute value function. Recall that the jump and link instruction is responsible for function calls. Jump and Link does two things. It first sets the address to return to after the function call in R31, and then it jumps to the function. So in this respect, the return address is in the R31 register and the jump register instruction must use R31 to return from the call. Hence Rs in the jump register instruction is equal to R31.

Knowledge Check 1.19

Answer A

The target address is given as 400400. Use the inverse of the formula in func2 to find DoImm

Answer B

Pay close attention to the inverse formula of func2. Note that that program counter is incremented by four. Also remember, shift right by 2 is the same as divide by four. And this division is the last operation of the inverse formula.

Answer C

Remember the first part of the formula of func2 is to do PC+4.

Answer D

That is correct. Using the inverse formula f12, we first do PC plus 4 subtract it from the target address. And then we either shift it right by 2 or divided by 4 to give us the immediate value.

MIPS64 Branch/Jump Instruction Uses 1.20

There is one more factor to consider. As we see in the high-level code, the absolute value function call must return the result to the main code. In our current MIPS mnemonic code, we have the result of the absolute value in the R4 register. If we recall, R4 is a register used to pass parameters to function calls. It thus cannot be used to return data from function calls. For that, we need to use either R2 or R3.

So in the final version of the code, we need to initialize R2 to X in order to keep the value in R2 in case X is already positive. We also need to make sure that the result of the subtraction that turns the negative value into a positive is also put into R2. This way, when the function call returns, the absolute value is in the R2 register and it can be used as the result variable in main code.

MIPS64 Branch/Jump Instruction Uses 1.21

Now let's take a look at an example of how jump and branch instructions are used for loops. Consider the following example. We have two integer variables X and Y, both greater than zero. The goal is to perform X times Y, without using the multiplication instruction.

One way to implement this function in a high-level C or C++ code is as follows. We have a while loop that uses a decrementing counter initialized at Y. Inside the loop, we add X to an accumulator as long as the counter is not zero. Note that for this approach, it doesn't matter if C and Y are always greater than 0. Another approach is to leverage the fact that X and yYare both greater than zero. In that case, we know that the loop will be executed at least once. So we can use a do while loop for the code. We will use that second approach to implement the MIPS mnemonic code.

MIPS64 Branch/Jump Instruction Uses 1.22

So let's detail the MIPS mnemonic code for our X times Y. Let us assume that X and Y are in R4 and R5 respectively and that the code segment starts at address 4D0.

If we look at the high-level code, we see that the first two operations initialize the counter to Y and set the accumulator R E S to zero. We can do the same for our MIPS code by initializing the register R8 to zero and the register R9 to the value of Y. In our MIPS code, we will thus use R8 for the accumulator and R9 for the loop counter. Since we have a do while loop, we know that we are going to add to the accumulator and decrement the loop counter at least once. To add to the accumulator, we simply use the doubleword ADD, or D ADD, R8, R8, R4 instruction to keep adding X to itself. To decrement the counter, we use the immediate add instruction D ADD I, where the immediate value is negative 1.

The next part of the code is to check if the counter is zero in order to determine if we need to get out of the loop or not. There are several ways to do this. We can use a branch if not equal to zero instruction, a branch if equal to zero, a branch if not equal, and probably other approaches. We choose to use branch if not equal. Since we need to compare the loop counter to zero, we use R9, which holds the loop counter, and R0, which is always zero, as the registers of our BNE instruction. If the counter is not yet zero, the branch is taken, and we need to go back to the two instructions that add the accumulator and decremental loop counter. This allows us to iterate through the loop one more time.

As we discussed before, the sequential instruction after a branch instruction must handle the case when the branch is not taken. For our code, the branch is not taken when the counter is equal to zero and we need to return from the function call. But instead of simply handling the return, we first need to store the final result of the accumulator in R2 or R3 register so that it can be available as a return value from the function call. So here we use the R2 register to return the value of the accumulator. This is similar to what we did before for the absolute value function. The last instruction handles the return from the function call and for that we use the jump register instruction. Assuming that the function would be called using jump and link, we know the return address will be in R31 register.

MIPS64 Branch/Jump Instruction Uses 1.23

We still need to determine what is the value of the do immediate parameter in our branch if not equal instruction. For that, we use the reverse of the func2 formula to get Do Immediate sign extended. We use the same approach as before the get the inverse of func2 and use the inverse to get the value Do Immediate. We observe that this Do Immediate is a negative value. Looking back at the complete code, we can see why Do Immediate is negative. This is because if the branch is taken, we go back up in the code. When the value is positive, as it was in the previous example, the branch was taken; we went down in the code.

MIPS64 Branch/Jump Instruction Uses 1.24

Here we have the MIPS mnemonic code for the other high-level code that works whether X or Y is zero. In this code, we also initiate registers R8 and R9 to hold the accumulator and the loop counter. The next stage is to implement the while loop. For that we use a branch if not equal instruction. If the branch condition is true, we go inside the loop to decrement the counter and to add to the accumulator.

If the loop condition is false, then we need to quit to loop. The way we do that in this case is by using the Jump instruction. This allows us to skip the instructions that are within the loop. We also see that there is a jump instruction after the instructions within the loop. This second jump instruction targets the branch if not equal instruction so we can go back and check the condition of the loop before we quit the function. The last two instructions are similar to the other code as they are responsible for copying the value of the accumulator in the R2 register and for returning from the function call.

Topics Covered 1.25

We have covered branch and jump instructions, which are fundamental in all instruction set architectures. As we have seen, these instructions allow the CPU to manipulate loops, function calls, and if and else conditions. This concludes the architecture portion of the course. Next, we will start discussing computer organization to understand how the CPU is designed to execute its instructions.