Week 13-14 Module 21, Part 1 – Computer Organization

(MIPS 64 Instructions)

Outline 1.2

Hi everyone. Today we will start discussing the computer organization section of this course. We will learn about the innerworkings of the CPU that allows it to execute its instructions and we will discuss different methods to optimize CPU performance.

So this is the outline for today. We will go over instruction types and formats, which are critical in determining how instructions are executed. We will then detail the execution steps for a small processor that we will design. We will then discuss processor clock rate to estimate how fast our processor is, and go over processor pipelining, its benefits, and the issues it presents.

MIPS64 Instructions 1.3

Let us discuss how instructions are encoded and formatted in MIPS64. First, let's go through basic characteristics of the layout of an instruction. Recall that each instruction in MIPS64 is 32 bits. Each instruction has an opcode. The opcode is 6 bits long and it is the most significant bits of the instruction. The role of the opcode is to indicate what operation the instruction performs. For example, the load double instruction has its own opcode and the store double instruction has its own opcode. Some instructions have the same opcode for formatting purposes. In such cases, the instructions have second opcodes also known as functions. These second opcodes play the role of indicating the operations of the instructions. In addition to an opcode, each instruction has one or many operands. The goal of the operands is to indicate where the data needed by the instruction is located. An operand can be a register number or an immediate value that is hardcoded in the instruction.

MIPS64 defines three formats for its instructions: register format, or R-format, Immediate format or I-format, and Jump format or J-format. The purpose of instruction formatting is to allow the processor to understand the role of the instruction and what operands it needs. This provides a compact method to design the processor. If each instruction was allowed to have its own format, then the CPU would have to be very complicated to handle any formatting approach.

MIPS64 Instruction Type: R-Format 1.4

R-format instructions have only registers as operands. An R-format can have between one and three registers and one of the registers can be a destination register. Another interesting aspect of R-format instructions is that that they all have the same opcode of 0. Therefore, each R-format instruction needs a second opcode, or a function, to detail its unique operation

The figure illustrates the layout of R-format instructions. The opcode for each R-format is the same. The instruction has three 5-bit fields for at most three registers, Rs, Rt, and Rd respectively. Another field of R-format instructions is the shift amount field. This field is only valid when the instruction is shift left logical or shift right logical. Otherwise, the shift mount field is zero. The last field of the instruction is the 6-bit second opcode or function. As we see, when we add the size of each field, we get 32 bits.

Let us illustrate how the processor determines what to do when it fetches an R-format instruction. Consider the example of this figure. The processor first reads the opcode and it sees it is all zeroes, meaning it is an R-format. The processor then reads the function field, which is the least significant 6 bits, to determine the operation of that R-format instruction. The function 101100 is for the D ADD instruction. The MIPS64 manual details the function field for each R-format instruction. Once the CPU knows the instruction is a doubleword ADD, it knows that it needs the operands Rs, Rt and Rd. Moreover, since it is not a shift instruction, the shift amount field is not used. The processor then converts the register numbers for the operands of the instruction and can form the complete instruction.

Knowledge Check 1.5 (Slide 9)

Answer A

Actually, each register can hold 64 bits of data. This is why we can do instructions such as double word add, where we add two 64-bit data.

Answer B

Actually, each register can hold 64 bits of data. This is why we can do instructions such as double word add, where we add two 64-bit data.

Answer C

That is correct. Since there are 32 possible labels for a register, they number of bit you need to represent a register field is log base 2 of 32, which is 5.

Answer D

Since each instruction is restricted to be 32 bits, the register field cannot be of unlimited width.

MIPS64 Instruction Type: I-Format 1.6

An I-format instruction has a 16-bit immediate value as an operand. An I-format instruction can also have up to two registers, Rs and Rt. Rt itself can be a source or a destination registers, whereas Rs can only be a source register. Unlike R-format instructions, each I-format instruction has a unique opcode. Therefore, there is no need for a second opcode.

Let us see how the processor breaks down an I-format instruction. Consider the instruction above. The processor first reads its opcode. The opcode matches to the double word add immediate instruction, which is an immediate instruction or an I-format instruction. Since the CPU knows that D ADD I uses two registers Rs and Rt, it

reads the two fields for Rs and Rt to determine the register numbers. The CPU then reads the 16-bit immediate value. Once that is done, the CPU has all of the operands of the instruction and can go onto the next step.

MIPS64 Instruction Type: J-Format 1.7

A J-format instruction only has one operand: a 26-bit immediate value. J-format instructions are used for jump instructions. Therefore, the 26-bit immediate value is used as an address parameter to use to calculate the target address of the jump instruction. Just like I-format, each J-format instruction has its own opcode and does not need a 2nd opcode. Just like all instructions, the CPU first read the opcode to determine the format. In the case of this example, the opcode matches to the Jump instruction. Since the jump instruction is a J-format, the CPU just reads the next 26 bits to determine the value of the address field.

Knowledge Check 1.8 (Slide 16)

Answer A

Remember, the only operand of a J-format instruction is the 26-bit address. Jump Register doesn't have such operand.

Answer B

The I-format instruction doesn't store the immediate value in a register. Instead, the immediate value is hardcoded into the instruction itself.

Answer C

That is correct. The Jump Register instruction uses one only the operand Rs, which is a register. Therefore, it is an R-format instruction

(Let's Build a Processor)

Instruction Execution Steps 2.1

OK. Now, we are going to the steps the processor go through when executing instructions. We will use the example of building a processor to illustrate these steps. Our processor is based on MIPS64 and has only 7 instructions. Our CPU can execute four R-format instructions: D ADD, D SUB, D OR and D AND. It can execute Load double and store double, as well as branch if equal to zero.

When executing an instruction, a CPU generally goes through the following steps: instruction fetch, instruction decode, instruction execute, memory, and WriteBack. During instruction fetch, the CPU reads the instruction from memory to determine what it is. The CPU uses the program counter to determine the address of the instruction in memory. During decode, the CPU determines the format of the fetched instruction, the operation the instruction is supposed to do, and gets its operands. The execute step performs the operation of the instruction. The processor goes through a memory step if the instruction needs to do a data memory access. For example, a store instruction

would go through the memory step. The processor goes through a WriteBack step if the instruction needs to write the result of its operation in a general purpose register. For example, the DADD instruction needs to write the result of its addition to the destination register and must go through the WriteBack step. As we can see, an instruction may not go through all the steps we have described. However, each instruction goes through the fetch decode and execute stages.

Knowledge Check 2.2 (Slide 19)

Answer A

That is correct. The load double instruction needs to accesss the memory to get the data and write the data to the destination register. The first part is done in the memory step, and the second one is done in the WriteBack step.

Answer B

The WriteBack step is only used to write to general-purpose registers R0-R31. The program counter is not a general-purpose register.

Answer C

The immediate value of DADDI is hardcoded in the instruction itself. So there is no need to go to memory to get it.

Answer D

The memory step is used to access data. Memory accesses for instructions are done at the fetch instruction fetch step. Moreover the BEQ instruction only updates the program counter.

Answer E

The JR reads the value of R31. The writeback step is used to write to general-purpose registers

Instruction Fetch 2.3

Let us discuss each execution step in detail. In the instruction fetch step, the processor uses the program counter to access the memory and get the instruction to execute. The syntax M and PC in brackets indicates a memory access where PC is the address is used for the access. When the memory returns the instruction, the CPU puts it in a special register known as the instruction register, or IR. This register always holds the 32 bits of the instruction being executed. The last part of the instruction fetch is to update the program counter to point to the next instruction. This is done by adding four to the current program counter. The value four is used to increment the program counter because each instruction is 32 bits or four bytes. Therefore, the CPU has to jump four bytes of address space to go point to the next instruction.

Instruction Decode 2.4

During the decode step, the CPU determines the operation that the instruction needs to perform, obtains the register values that may be needed, and allocates the functional resources that the instruction needs for its operation. For the purposes of our CPU, we

only need one functional unit: an ALU. This is because our CPU does not have complex instructions. The same ALU is used for all operations that the CPU needs to perform. This ALU has two outputs ALUOut1 and ALUOut2. Here is a little bit mor e detail on the decode step for our CPU. During that step, our CPU automatically accesses the registers for the Rs and Rt operands and sign-extends the 16-bit immediate value, regardless of the instruction type.

The CPU then puts the value of Rs in a special register called A, It puts the value of Rt into a special register called B, and it puts the sign-extended value into a special register called I MM, or immediate. Note that A, B, and I MM, are special registers like PC and IR and cannot be accessed by the code. They are only used to store temporary parameters that the CPU needs to complete the instruction.

Knowledge Check 2.5 (Slide 25)

Answer A

Remember, during the fetch step, the CPU only increments the program counter. It cannot perform anything of relevance for the instruction because it will known what the instruction is until the decode step.

Answer B

That is correct. The fetch step only increments the counter by 4, under the assumption that the next instruction to be executed is the sequential one. The CPU doesn't yet know if the instruction is a branch. That is what the decode step is for.

Answer C

Here you used the wrong formula to calculate the target address. Remember, the first part of the formula is to increment PC by 4. But remember, for this question, the CPU just completed the fetch step. So it doesn't yet know that it is doing a branch instruction.

Answer D

Keep in mind the operations that the CPU performs during the fetch step. It first sends the value of the program counter to get the instruction, then it increments the program counter by four to point to the next instruction in the code sequence

Instruction Execute 2.6

Regardless of the instruction, the processor goes through the same procedures for fetch and decodes steps. Things begin to change when we get to the execute step. This is because each instruction has a different role. As a result, we have to look at the execute step for each instruction.

Let us first look at the load double and store double instructions. During the execute step, the CPU calculates the memory address that either the load or the store instruction needs. Recall that both load and store use the Rs register to hold the base

address of the data segment, and use the 16-bit do immediate as the offset within the data segment. Also, remember that to calculate the address, the CPU needs to first sign extend the do immediate, then add it to the base address of the data segment. During the decode step, the CPU had already moved the value of Rs to A and had already sign-extended the do immediate and had put into I MM. Therefore, during the execute step for load double or store double instruction, the CPU just adds A and I MM to get the address of the memory access. The result of the addition is stored in ALUOut1.

Instruction Execute 2.7

For the arithmetic and logic R-format instructions, the ALU of the CPU simply performs the operation using the values of the registers Rs and Rt. Recall that during the decode step, the CPU already has Rs and Rt in A and B respectively. The only thing that is an issue is how can the ALU determine what operation to perform for that R-format instruction. Recall the second opcode or function field of the R-format instruction. The ALU has a lookup table that contains the arithmetic or logic operation to perform based on the value of the function field. So, the CPU simply sends the value of A and B registers and the value of the function field of the R-format instruction to the ALU. The latter used its table to perform the operation and stores the result in ALU Out 2.

Instruction Execute 2.8

For the branch if equal to zero instruction, the CPU must perform several operations during the execute step. It must calculate the target address of the branch instruction, check the branch condition, and update the program counter.

Let us first see how the CPU calculates the target address for the branch instruction Remember the formula for func2 that is used to calculate the target address. During the fetch step, the CPU already incremented the program counter by 4, and during the decode step, the CPU sign-extends the 16-bit immediate value. Those two calculations from fetch and decode steps are needed to calculate the target address. Therefore, during the execute step, the CPU can simply shift the sign-extended immediate value and add it to PC to calculate the target address. This target address is put in ALUOut1. The next operation is to check the condition of the branch if equal to zero. To check if a value is equal to zero, the CPU used the ALU to subtract that value with zero. The ALU has what is called a zero flag. That is a one-bit output that is set to one when the result of its operation is zero.

So, to check the condition, the ALU used the BranchOp operation, which is a subtraction of the Rs, which is in the A, and R0, which holds 0. If the subtraction is zero, then the zero flag of the ALU is set to one. The last operation is to update the program counter if the branch condition is true. For this, the CPU simply writes the target address to the program counter if the value of the zero flag is one. Otherwise, the program counter remains its original value from the fetch step where it was incremented by 4. It is important to realize that although the branch if equal to zero instruction writes to the

program counter register, this is not done at the WriteBack step. This is because the WriteBack step is only for general purpose registers R0 through R31. The program counter itself is not a general-purpose register.

Knowledge Check 2.9 (Slide 37)

Answer A

Remember: A is just a register within the CPU that holds the value of Rs. And Zero is a single-bit flag that is raised to one when the result of an ALU operation is zero.

Answer B

Remember: A is just a register within the CPU that holds the value of Rs. And Zero is a single-bit flag that is raised to one when the result of an ALU operation is zero.

Answer C

That is correct. A is a temporary register that holds the value of RS, which in our case is FFFC. 0 is a 1 bit flag that holds the result of the operation of the ALU. Since the operation for this instruction is RS minus 0, the result is not 0. Therefore, the zero flag is remaining 0.

Answer D

Remember, the value of the Zero flag is raised to one when the result of the last ALU operation is one. The last ALU operation performed was for the comparison of R5 and R0 for the BEQZ instruction.

Instruction Memory 2.10

Let us now look at the memory step. This step is only used by load and store double instructions in our CPU. For the load double instruction, the CPU sends the memory address to the memory controller in order to obtain the read the data. Since the address of the data was already calculated during the execute step, the CPU can simply make the memory request using the address stored in ALUOut1. When the data is returned from memory, the CPU puts it in the load memory double or LMD register. LMD is a special purpose register that temporarily holds the value of data obtained from memory loads.

For the store double instruction, the CPU must write the data from the register Rt to the memory using the address calculated in the execute step. Since Rt is put in the B temporary register during decode stage, the memory step is fairly straightforward for this instruction.

The WriteBack step is only useful for the Load double and the R-format arithmetic and logic operations. This is because only these instructions of our CPU need to write to general-purpose registers. For load double, the CPU takes the content of LMD and writes it into the Rt register. For the R-format instructions, the CPU needs to write the result of the ALU operation in the Rd register. Since the result of the ALU operation was

put in ALUOut2 during execute step, the CPU can simply use the content of ALUOut2 to put in Rd.

High-Level Diagram 2.11

Ok. Now, let us design a high-level state diagram of our small CPU. The high-level diagram will break down the steps to execute each instruction into states and will show when to transition from one state to the other.

High-Level Diagram 2.12

The figure here shows the state diagram of our CPU. Each state in the diagram has a unique numerical ID, where the first state has the numerical ID zero. Each state encompasses an execution step of the processor for a given operation. State zero and state one of the diagram are common for all instructions of our CPU because the same operations are performed for fetch and decode steps. The diagram starts diverging at the execution step because that is when the operations vary according to the specifics of the instruction. We see that from state 1, many arrows in the diagram that transition from one state to another and they are labeled by specific instructions. This is to indicate which instructions need those states at the end of the arrows. For example, coming from state 1, if the CPU is executing a Load double instruction it has to go to state 2.

One important thing to note is that the last state of each instruction always transitions to the state zero, which is the instruction fetch state. For example, the last state for the Load double instruction is state 4. Once that is complete, the CPU goes to state 0, which. This allows the processor to execute the next instruction upon completion of the current one.

(Processor Pipelining)

Processor Clock Rate 3.1

Let us now discuss the performance of this CPU. First, let's go over the concept of CPU clock rate. Usually, when we buy a computer, one of the factors that we look at is the CPU frequency. The frequency essentially is representative of the clock rate, which defines how fast each operation can be performed. The metric of the clock rate is the clock cycle. To determine out how many seconds a clock cycle last, we simply take the inverse of the CPU frequency. For example, if we have a 1 gigahertz CPU, one clock cycle takes 1 divided by 1 gigahertz which equals 1 Nano second. What this means is that each operation of the CPU takes 1 nano second to perform. In this respect, an operation can be thought of as the work of one state in our CPU high-level diagram.

Processor Clock Rate 3.2

So if we know that each state takes one clock cycle, to find out how many states it takes to execute a given instruction. Once can simply add up all the states it goes through. Consider for example the load double instruction. Like all other instructions, Load

double goes through states 0 and 1 for fetch and decode. Then load double goes to state 2 of the diagram for execute, state 3 for memory, and state 4 for WriteBack. The instruction thus goes to 5 states and thus takes 5 clock cycles. Using this approach, we can calculate how long it would take for our CPU to execute a given code. For example, using this A = B or C code, we use the high-level state diagram to calculate the number of cycles each instruction, then add the number of cycles of the instructions together, and we estimate that our CPU takes 18 cycles to complete this code.

So, how fast is this CPU? Well, considering the fact that modern low-end CPUs such as the ones used in smart phones can execute at least 4 instructions per cycle, our design is not very fast. This is primarily because in our current design, the CPU can execute only one instruction at a time.

Processor Performance 3.3

Let us consider the following example. Given two instructions load double and double word add, our CPU will do the following. It will first execute the load double instruction. That is, it will go to states 0, 1, 2 3, and 4, requiring 5 cycles to complete. Then the CPU transitions to state zero to start executing the double word add instruction. Since doubleword add takes 4 clock cycles, the two instructions require 9 cycles to complete. This type of CPU that can only execute one instruction at a time is called an unpipelined CPU. The opposite is called a pipelined CPU.

Processor Performance 3.4

So here are several methods that CPU designers use to optimize the performance. For this course, we'll focus on pipelining. If you want to find out more information about the other methods, click on the following links.

Processor Pipelining 3.5

Let us discuss pipelining in more detail. When a CPU is pipelined, it breaks down each execution step into a stage, where each can be used independently by an instruction. In addition, each stage uses buffers, commonly known as latches, to obtain information about what the instruction did in the previous stage. So let's use this example to illustrate the benefits of pipelining. When one instruction starts executing, it first goes in the execution, instruction fetch stage in the pipeline. Just like the states in the high-level diagram of the CPU, each pipeline stage takes one clock cycle to complete.

When the instruction completes the fetch stage, it goes to the decode stage of the pipeline in cycle 2. This means the fetch stage is free at that cycle because no instruction is currently using it. The pipelined CPU can thus send another instruction in the fetch stage to begin its execution. At this clock cycle 2, there are multiple instructions being executed at the same time. Hence we have what is called instruction level parallelism or ILP. At clock cycle 3, the CPU sends the first instruction to the execute stage, the second instruction to the decode stage, and the third instruction to

the fetch stage. Using this approach, the CPU can have one instruction being executed in each stage at all times. However, note that we cannot have multiple instructions in the same stage at the same time. This is done only by Superscalar CPUs. Right now, we only have a pipelined CPU. Using pipelining method, our code can take 7 cycles instead of 18 cycles, doubling its performance.

One interesting note about pipelining is that some instructions may need to go to pipeline stages that they don't use. For example, the double word instruction doesn't need the memory stage, because it doesn't read or write data. But since, it must go to the writeback stage to write the result in the register, it has to go through the Memory. This is because in the pipelined design, the CPU simply moves an instruction from one stage of the pipeline to the next stage until it completes.

Hazards 3.6

The main issues with a pipelined CPU are conflicts that it encounters when executing multiple instructions simultaneously. These conflicts are commonly known as hazards and are categorized in three groups: structural hazards, data hazards, and branch hazards. Structural hazards occur when multiple instructions need to access the same CPU resource at the same time. Data hazards occur when an instruction needs to read a data that is being modified by another instruction. And branch hazards occur when a branch instruction is being executed and the CPU needs to know if the branch is taken or not before it fetches the next instruction.

Structural Hazards 3.7

Let us take a look at structural hazards in more detail. Consider the code here. Remember that in our CPU, there is only one ALU. That same ALU is used by the fetch stage of the pipeline to increment the program counter by four and is used by the execute stage for operations. In this example, at clock cycle 3, the first instruction is in the execute stage and the third instruction is in the fetch stage. Therefore, both instructions need to use the ALU at the same cycle. Here is another example of structural hazard. Let us assume our CPU has only one memory port. The same port is used by the instruction fetch stage and by the memory stage. In this example, the first instruction needs to use the memory port at cycle 4 in order to load the data from memory and the fourth instruction needs to use the port at cycle 4 to fetch the instruction.

The main way to mitigate structural hazards is by replicating resources. Therefore, we can add two ALUs in our CPU, one for the instruction fetch stage to increment the counter, the program counter. And another in the execute stage to perform the functionalities of the instructions. Note that structural hazards rarely occur in modern processors because they have several replications of resources for different stages.

Data Hazards 3.8

Data hazards are relevant for instructions that have dependency issues. Let us use this example to illustrate such dependencies. At clock cycle 5, the third instruction reads the value of R10 because it needs it for its double word OR instruction. However, the correct value of R10 is written at clock cycle 6 by the second instruction, that correct data is not available when the OR instruction needs at cycle 5. This is known as a read after write data hazard.

Note that although we use data in register to illustrate data hazards, they can also use for data that is in memory. One way to overcome data hazards is via data forwarding. The basic idea of forwarding is to send data to the execute stage as soon it is available in the CPU. Using our example, the value of R10 is available in the CPU at clock cycle 5, because that is when the memory accesses and puts it in the load memory data register. Therefore, the CPU can forward the data from that load memory data register to the ALU in the execute stage to guarantee that the correct value of R10 is used for the doubleword OR operation.

Another approach to mitigate data hazards is by instruction re-ordering. The basic idea here is for the compiler to find independent instructions to put between dependent ones to avoid data hazards. This can be done as long as the re-ordering does not modify the correctness of the code. Consider the following example. The instruction at address C8 depends on the instruction at address C4 because the instruction at address C4 first writes to R10 then the one at C8 reads R10. We also observe that the two instructions at addresses D0 and D4 did not depend on any of the instructions between C4 and D0. Therefore, the result of the previous instructions will not affect them if we move it up in the code. We can thus move the independent instructions at D0 and D4 in between the dependent ones. This creates two clock cycles extra between the instruction that writes to R10 and the one that reads from R10. Therefore, when the instruction that reads R10 is doing so at the decode stage, R10 is at least being written to at that same cycle.

We know that since the compiler is responsible for doing this re-ordering work, the addresses of the instructions that are re-ordered are modified to reflect where they are in the code. We also note that this instruction re-ordering is not the same as out-of-order execution. Out-of-order execution is an additional re-ordering that is done dynamically by the processor, not by the compiler.

Branch Hazards 3.9

Branch hazards occur when there is a branch instruction the CPU needs to fetch the next instruction in the pipeline. Since the result of the branch condition and the target address of the branch will not be available until the execute stage, the CPU does not what to fetch. Consider the example here. At cycle 4, the CPU fetches the instruction at address E0. In the next cycle, that instruction moves to the decode stage. To keep the pipeline busy, the CPU needs to fetch a new instruction at that same cycle. However,

the CPU does not yet know if the branch is taken or what that target address is in that case. Although we have described the branch hazards for conditional branches, the same issue occurs for unconditional jump instructions because the target address is calculated at the execute stage of the pipeline.

There are several ways to overcome branch hazards. The most naïve way is to simply stall the CPU execution until the branch instruction goes to the execute stage to calculate the target address. This is known as a pipeline bubble. In fact, pipeline bubble can be used to mitigate just about every hazard. The issue with pipeline bubble is that it does not optimize the use of the pipeline. If we have to always stall due to hazards, then the benefits of pipelining are limited. Another approach to mitigate branch hazards is for the compiler to add a branch delay slot after each branch instruction. A branch delay slot is essentially an instruction within the program code that can be executed whether the branch is taken or not. This is similar to the instruction reordering approach.

The most common approach to mitigate branch hazards, however, is by using branch prediction. The basic idea here is to predict if the branch is taken or not based on past history. The CPU has a table to keep the history of each branch instruction. For example, for each branch instruction, the table holds its last target address and whether the branch was taken the last few times it was executed. The CPU then uses that pattern of the last few times the branch was executed to predict if it will be taken and to provide the CPU the target address at the fetch stage. Branch predictors are over 99% accurate in modern processors.

Knowledge Check 3.10 (Slide 59)

Answer A

Remember. Since only one instruction uses the CPU at a time, we don't get resource conflicts or structural hazards in a unpipelined CPU. Moreover, since each instruction completes before the next one is fetched, we don’t get dependency issues or data hazards, as well as branch taken/not taken issues or branch hazards. So there are no branch hazards in an unpipelined CPU.

Answer B

That is correct. Pipelining only allows one instruction in a pipeline stage at a time. Therefore, we can't have 2 instructions in the decode stage at the same time.

Answer C

Remember, in an unpipelined CPU, the instruction must complete all of its steps before the CPU fetches the next one. Therefore, only one instruction can use the CPU at a time.

Answer D

To maximize the throughput of the pipelined CPU, it must always have a new instruction in the fetch stage. The only way that occurs is if the instructions in

the other stages are moving along. This way, there is an instruction at each stage of the pipeline at all times.

Topics Covered 3.11

We covered the various instruction formats of MIPS64 architecture and detailed how they are used to encode information about the operations of different instructions. We also discussed the five main stages of CPU execution and their intricacies. Then, we covered processor pipelining, which is one of the most fundamental aspects of computer organization. We discussed the benefits of pipelining in term of throughput, the hazards they present, and the common mechanisms used to overcome these hazards.