MITOCW | MIT6_004S17_15-02-01_300k

In this lecture, we're going to use the circuit pipelining techniques we learned in Part 1 of the course to improve the performance of the 32-bit Beta CPU design we developed in Part 2 of the course.

This CPU design executes one Beta instruction per clock cycle.

Hopefully you remember the design!

If not, you might find it worthwhile to review Lecture 13, Building the Beta, from Part 2.

At the beginning of the clock cycle, this circuit loads a new value into the program counter, which is then sent to main memory as the address of the instruction to be executed this cycle.

When the 32-bit word containing the binary encoding of the instruction is returned by the memory, the opcode field is decoded by the control logic to determine the control signals for the rest of the data path.

The operands are read from the register file and routed to the ALU to perform the desired operation.

For memory operations, the output of the ALU serves as the memory address and, in the case of load instructions, the main memory supplies the data to be written into the register file at the end of the cycle.

PC+4 and ALU values can also be written to the register file.

The clock period of the Beta is determined by the cumulative delay through all the components involved in instruction execution.

Today's question is: how can we make this faster?

We can characterize the time spent executing a program as the product of three terms.

The first term is the total number of instructions executed.

Since the program usually contains loops and procedure calls, many of the encoded instructions will be executed many times.

We want the total count of instructions executed, not the static size of the program as measured by the number of encoded instructions in memory.

The second term is the average number of clock cycles it takes to execute a single instruction.

And the third term is the duration of a single clock cycle.

As CPU designers it's the last two terms which are under our control: the cycles per instruction (CPI) and the clock

period (t_CLK).

To affect the first term, we would need to change the ISA or write a better compiler!

Our design for the Beta was able to execute every instruction in a single clock cycle, so our CPI is 1.

As we discussed in the previous slide, t_CLK is determined by the longest path through the Beta circuitry.

For example, consider the execution of an OP-class instruction, which involves two register operands and an ALU operation.

The arrow shows all the components that are involved in the execution of the instruction.

Aside from a few muxes, the main memory, register file, and ALU must all have time to do their thing.

The worst-case execution time is for the LD instruction.

In one clock cycle we need to fetch the instruction from main memory (t_IFETCH), read the operands from the register file (t_RF), perform the address addition in the ALU (t_ALU), read the requested location from main memory (t_MEM), and finally write the memory data to the destination register (t_WB).

The component delays add up and the result is a fairly long clock period and hence it will take a long time to run the program.

And our two example execution paths illustrate another issue: we're forced to choose the clock period to accommodate the worst-case execution time, even though we may be able to execute some instructions faster since their execution path through the circuitry is shorter.

We're making all the instructions slower just because there's one instruction that has a long critical path.

So why not have simple instructions execute in one clock cycle and more complex instructions take multiple cycles instead of forcing all instructions to execute in a single, long clock cycle?

As we'll see in the next few slides, we have a good answer to this question, one that will allow us to execute *all* instructions with a short clock period.

We're going to use pipelining to address these issues.

We're going to divide the execution of an instruction into a sequence of steps, where each step is performed in successive stages of the pipeline.

So it will take multiple clock cycles to execute an instruction as it travels through the stages of the execution pipeline.

But since there are only one or two components in each stage of the pipeline, the clock period can be much shorter and the throughput of the CPU can be much higher.

The increased throughput is the result of overlapping the execution of consecutive instructions.

At any given time, there will be multiple instructions in the CPU, each at a different stage of its execution.

The time to execute all the steps for a particular instruction (i.e., the instruction latency) may be somewhat higher than in our unpipelined implementation.

But we will finish the last step of executing some instruction in each clock cycle, so the instruction throughput is 1 per clock cycle.

And since the clock cycle of our pipelined CPU is quite a bit shorter, the instruction throughput is quite a bit higher.

All this sounds great, but, not surprisingly, there are few issues we'll have to deal with.

There are many ways to pipeline the execution of an instruction.

We're going to look at the design of the classic 5-stage instruction execution pipeline, which was widely used in the integrated circuit CPU designs of the 1980's.

The 5 pipeline stages correspond to the steps of executing an instruction in a von-Neumann stored-program architecture.

The first stage (IF) is responsible for fetching the binary-encoded instruction from the main memory location indicated by the program counter.

The 32-bit instruction is passed to the register file stage (RF) where the required register operands are read from the register file.

The operand values are passed to the ALU stage (ALU), which performs the requested operation.

The memory stage (MEM) performs the second access to main memory to read or write the data for LD, LDR, or ST instructions, using the value from the ALU stage as the memory address.

For load instructions, the output of the MEM stage is the read data from main memory.

For all other instructions, the output of the MEM stage is simply the value from the ALU stage.

In the final write-back stage (WB), the result from the earlier stages is written to the destination register in the register file.

Looking at the execution path from the previous slide, we see that each of the main components of the unpipelined design is now in its own pipeline stage.

So the clock period will now be determined by the slowest of these components.

Having divided instruction execution into five stages, would we expect the clock period to be one fifth of its original value?

Well, that would only happen if we were able to divide the execution so that each stage performed exactly one fifth of the total work.

In real life, the major components have somewhat different latencies, so the improvement in instruction throughput will be a little less than the factor of 5 a perfect 5-stage pipeline could achieve.

If we have a slow component, e.g., the ALU, we might choose to pipeline that component into further stages, or, interleave multiple ALUs to achieve the same effect.

But for this lecture, we'll go with the 5-stage pipeline described above.

So why isn't this a 20-minute lecture?

After all we know how pipeline combinational circuits: We can build a valid k-stage pipeline by drawing k contours across the circuit diagram and adding a pipeline register wherever a contour crosses a signal.

What's the big deal here?

Well, is this circuit combinational?

No!

There's state in the registers and memories.

This means that the result of executing a given instruction may depend on the results from earlier instructions.

There are loops in the circuit where data from later pipeline stages affects the execution of earlier pipeline stages.

For example, the write to the register file at the end of the WB stage will change values read from the register file in the RF stage.

In other words, there are execution dependencies between instructions and these dependencies will need to be taken into account when we're trying to pipeline instruction execution.

We'll be addressing these issues as we examine the operation of our execution pipeline.

Sometimes execution of a given instruction will depend on the results of executing a previous instruction.

Two are two types of problematic dependencies.

The first, termed a data hazard, occurs when the execution of the current instruction depends on data produced by an earlier instruction.

For example, an instruction that reads R0 will depend on the execution of an earlier instruction that wrote R0.

The second, termed a control hazard, occurs when a branch, jump, or exception changes the order of execution.

For example, the choice of which instruction to execute after a BNE depends on whether the branch is taken or not.

Instruction execution triggers a hazard when the instruction on which it depends is also in the pipeline, i.e., the earlier instruction hasn't finished execution!

We'll need to adjust execution in our pipeline to avoid these hazards.

Here's our plan of attack: We'll start by designing a 5-stage pipeline that works with sequences of instructions that don't trigger hazards, i.e., where instruction execution doesn't depend on earlier instructions still in the pipeline.

Then we'll fix our pipeline to deal correctly with data hazards.

And finally, we'll address control hazards.