Evan Smith

CSE 661, Adv Computer Arch, Abdallah

Homework 2

Pipelining and the Design of ODMIS

Pipelining is the baseline process that drives most modern computing systems. The ability to parallelize instruction computation is a fantastic way to dramatically increase the capabilities of a system. Here I will discuss points of the conventional 5-stage RISC pipeline in which my new demonstration "language" designed for this course, ODMIS, makes some interesting and noteworthy divergences from MIPS.

The first stage of the pipeline is the Fetch or read cycle. The instruction is fetched from the instruction memory location and loaded into the instruction register or similar. This step is obviously key to the rest of the process, as the entire processor must be fed instructions to continue operations. The general layout of a RISC instruction is led by an operation code (or opcodes), which indicates to the processor during the decoding step which flags and values should be populated before execution. These opcodes have a human-language name that corresponds to their function. Common examples of operations are Add, Subtract, Load, and Store. These human-language names are variable depending on the assembly language, and so is the format of the instruction itself.

The language that is used as the baseline in this course, MIPS, has one approach that splits instructions into 3 different types. Each type has its own structure to the following bits, divided into some number of data blocks that represent arguments, registers, immediate values, or other components. The type of the instruction is determined by its opcode, and so the processor cannot blindly load the instruction's data to a certain location without first decoding the instruction in the following step. There are other design choices that can be made at this step. For example, my created language for this assignment, Opcode-Driven Microprocessor Instruction Set (ODMIS) has blocks of data in the instruction that are consistent regardless of the actual operation specified.

Instead, the instruction consists of an opcode block of consistent length, followed by five blocks of consistent length. This would allow a potential architecture designed around the language to parse the arguments before decoding the instruction, and then leaving the opcode to drive the appropriate flags that determine the actions that the CPU takes on those arguments. This gave the "language" its name, since the opcode alone indicates the operational flags that should be set, as well as some other critical information that is discussed in the next section.

The second stage in the pipeline is the Decode step, in which the previously mentioned opcode is used to drive the behavior of the other components in the CPU. Examples of this control are enabling or disabling the read/write flags on registers, the program counter, accumulator (if present), among other things. This step can also change larger control data components, such as the active address in the register stack, or pass along immediate values if not handled during the Fetch step. The decode step, like all this pipeline's stages, can be described or imagined as having many different implementations ranging from simple to immensely complex. At its core, the processor must be able to look up the opcode and return a set of outgoing flags and values to set for its dependent components to act upon in the following cycle.

In the case of variable length argument blocks (like in MIPS) or even variable length instructions overall, the parsing of the opcode will also drive how the rest of the instruction is parsed and decoded. This is where the potential gains, however small, of the ODMIS concept lie. The decoder will still rely on the opcode to decide on operational flags, but the argument parsing can be completed synchronously with the opcode parsing. This allows a standard set of transistors to hold the components of the instruction, and simply just needs those blocks to be provided to the appropriate component (register stack, ALU, memory retrieval, etc.).

Our third stage is to Execute the instruction that was decoded. This stage is the most amorphous of all the stages since the term "execute" is completely depended on the opcode. The most common executing component is an Arithmetic Logic Unit (ALU) that can do logical operations on inputs and produce a set of outputs including several indicator flags. This is where instructions such as ADD, SUBTRACT, OR, AND, bit-shifts, and other such operations will take place.

While ODMIS doesn't inherently bring anything different to this stage vs MIPS, it is a good time to discuss the way that ODMIS treats numerical types. To fit in with the concept of static partitions in the instruction structure, numbers in ODMIS are defined as either integers or decimals. Both are stored as two arguments per value, allowing the instructions for integer and decimal operations to be syntactically similar. The difference is that the type of the arguments is indicated by the opcode itself, such that there is a distinct operation for adding ints vs decimals. While this is largely an academic exercise that will bloat the instruction set for each numeric type, it brings up an interesting capability that the same data register could be treated as different values depending on the instruction that references it.

In my implementation for this assignment, I chose to lock each memory location to a certain numerical type, if it was storing that information. To achieve this, I assigned one bit in each location to be an indicator, essentially "IS_INT". This concept mimicked the ideas of signed values that is widely used in other architectures. This can also be paired with the signed concept as well, allowing for negative values to be stored. Storing the type of the data in its storage location enables the user to be type-safe when executing code. If a decimal-opcode is invoked on an integer register, we can tell that this is an error at runtime and can prevent the execution of the miswritten instruction. There is, of course, flexibility on this point. For example, the processor could accept integer operations on decimal registers, with the caveat that only the whole component of the value would be considered. This is made easy by the argumentation separation described in ODMIS – we can simply take from only the first representative argument for the decimal value, which is defined explicitly as the whole number component.

The fourth and fifth stages, Memory and Write, can be combined for the purposes of discussing their impact from ODMIS's changes. These stages encompass the access of registers, memory, and cache to read and write values that are input or computed during the previous stages. This is the part of the instruction's lifecycle in which conflicts with preceding instructions can arise, which require previous planning to avoid unexpected results. The most naïve approach to solving simultaneous reference issues (for example, one register being modified and read from in the same cycle) is to inject stalls into the instruction stack, delaying the memory access by the offending command. As has been established in class, there are numerous other established techniques that allow the processor to more efficiently solve this issue, but unfortunately ODMIS doesn't provide any more clear structures with which to alleviate the problem. A change in the implementation of ODMIS is that there is no need to distinguish between floating-point and integer registers on the processor. As discussed above, a leading flag on the register indicates its type, and the structure of the data is always composed of two equal-sized arguments that either define a 2-unit long integer or a decimal with 1-unit of whole value and 1-unit of fractional value. This means that use of a common set registers to hold any possible values generated by the instructions is a totally reasonable and minimalistic approach to designing the on-chip register set. Of course, separation may still be desired for the specific operations of the CPU, but it is not required.

Overall, there is much that is admittedly impractical about the ODMIS language. Notable elements include a limited range of fractional representation and core instructional bloat. However, it was exciting to try and change some of the core premises of the MIPS language when designing a smaller, experimental variation. I particularly enjoy the potential improvements to the pipelining system in which the arguments could be passed to the processor or other components earlier on in the process due to their uniform design across all instructions. Additionally, the possibility of producing different results when operating on the same register value but with different value-type opcodes is a thought-provoking design element that could be utilized in some specialized situations. Otherwise, type-marking the storage locations themselves, as was done in my simulator, allows for imbedded run-time detection of mixed-type operations and can protect against unintended outcomes. All things considered, the practice of creating the logic of ODMIS led me to a greater appreciation of both the designers and innovators of MIPS as well as those that have iterated countless times to bring the state of computer architecture to the stage we are in now.