CHAPTER 11: REDUCED INSTRUCTION SET COMPUTING

RISC processors seek to improve performance by taking a “less is more” approach. They reduce the number of instructions in their instruction sets to minimize the time needed to process each instruction. Toward the end, they use fixed length instructions, fewer addressing modes, and only load and store instructions to access memory. They also have hardwired control units, and separate instruction and data streams.

The idea of RISC-labeled designs has been around for a while now, since 1975 or so. These original design aspects included the observations that the memory restricted compilers of the time were often unable to take advantage of features intended to facilitate manual assembly coding, and that complex addressing inherently takes many cycles to perform, due to the implied additional memory access. It was argued that such functions would better be performed by sequences of simpler instructions, if this could yield implementations simple enough to cope with really high frequencies, and small enough to leave room for many registers, factoring out slow memory accesses. Uniform, fixed length instructions with arithmetics restricted to registers were chosen to ease instruction pipelining in these simple designs, with special load-store instructions accessing memory.

For any given level of general performance, a RISC chip will typically have far fewer transistors dedicated to the core logic which originally allowed designers to increase the size of the register set and increase internal parallelism. Features typically found in RISC architecture include: uniform instruction formats, using a single word with the opcode in the same bit positions in every instruction, demanding less decoding; identical general purpose registers, allowing any register to be used in any context, simplifying compiler design; simple addressing modes, which is complex addressing performed via sequences of arithmetic and/or load-store operations; and fewer data types in the hardware. Some CISCs have byte string instructions, or support complex numbers; this is so far unlikely to be found on a RISC. RISC designs are also more likely to feature a Harvard memory model, where the instruction stream and the data stream are conceptually separated; this means that modifying the memory where code is held might not have any effect on the instructions executed by the processor (because the CPU has a separate instruction and data cache), at least until a special synchronization instruction is issued. This allows both caches to be accessed simultaneously, which can often improve performance. Many early RISC designs also shared the characteristics of having a branch delay slot. A branch delay slot is an instruction space immediately following a jump or branch. The instruction in this space is executed, whether or not the branch is taken (in other words the effect of the branch is delayed). This instruction keeps the arithmetic logic unit of the CPU busy for the extra time normally needed to perform a branch. Nowadays the branch delay slot is considered an unfortunate side effect of a particular strategy for implementing some RISC designs, and modern RISC designs generally do away with it

RISC processors use instruction pipelines to overlap instruction processing. Instruction pipelines are a technique used to increase throughput (the number of instructions that can be executed in a unit of time). The fundamental idea behind instruction pipelining is to split the processing of a computer instruction into a series of independent steps, with storage at the end of each step. This allows the computer’s control circuitry to issue instructions at the processing rate of the slowest step, which is much faster than the time needed to perform all steps at once. The term “pipeline” refers to the fact that each step is carrying data at once (like water), and each step is connected to the next (like the links of a pipe). The advantages of pipelining are that the cycle time of the processor is reduced, thus increasing instruction issue-rate in most cases. Some combinational circuits, such as adders or multipliers, can be made faster by adding more circuitry. If pipelining is used instead, it can save on circuitry by adding a more complex combinational circuit instead. The disadvantages of using pipelining include branch delays: a non-pipelined processor executes only a single instruction at a time. With pipelining, there are problems with serial instructions being executed concurrently. Consequently the design is simpler and cheaper to manufacture without pipelining. The instruction latency in a non-pipelined processor is slightly lower than in a pipelined equivalent. This is due to the fact that extra flip flops must be added to the data path of a pipelined processor. Also, a non-pipelined processor will have a stable instruction bandwidth. The performance of a pipelined processor is much harder to predict and may vary more widely between different programs. Processors resolve these problems using no-op insertion, instruction reordering, stalling, data forwarding, annulling, and branch prediction.

Most RISC processors include a large number of registers to reduce memory accesses, thus improving system performance. Processors generally have access to only a subset of these registers at any given time. They use register windowing or register renaming to coordinate access to the registers.

The use of register windows, another common feature in RISC design, is a technique to improve the performance of a particularly common operation, the procedure call. Although a RISC processor has many registers, it may not be able to access all of them at any given time. Most RISC CPUs have some global registers, which are always accessible. The remaining registers are windowed so that only subsets of the registers are accessible at any specific time. While registers are almost a universal solution to performance, they do have a drawback. Different parts of a computer program all use their own temporary values, and therefore compete for the use of the registers. Since a good understanding of the nature of program flow at runtime is very difficult, there is no easy way for the developer to know in advance how many registers they should use, and how many to leave aside for other parts of the program. In general these sorts of considerations are ignored, and the developers, and more likely, the compilers they use, attempt to use all the registers visible to them. This is where register windows become useful. Since very part of a program wants registers for its own use, it makes sense to provide several sets of registers for the different parts of the program. Of course, if these registers were visible, there would simply be more registers to compete over; the “trick” is to make the invisible. This is actually somewhat simpler than it might sound; the movement from one part of the program to another during the procedure call is easily “seen”, it is accomplished by one of a small number of instructions and ends with one of a similarly small set. In Berkeley RISC design, only eight registers were visible to the programs, out of a total of 64. The complete set of registers was known as the register file, and any particular set of eight as a window. Overlapping register windows provide an efficient method for transferring parameters and results between a program and its subroutines.