Week 7: RISC vs CISC

Learning Objectives (Dr. Feister edits):

- **Compare two broad categories of computer architectures**
  - computer instruction set architecture, RISC and CISC. CISC has a wide variety of different instructions and addressing modes, whereas RISC keeps the instruction set small and relatively simple. These have various tradeoffs in terms of computer performance as well as complex effects on compiler creation and program execution

RISC

↳ Characteristics

↳ **One instruction per cycle**

↳A machine cycle is defined to be the time it takes to fetch two operands from two registers, perform an ALU operation, and store the result in a register. Essentially the computing time it took to perform any operation from the last lab. Microcode manipulates the control registers with a series of coded steps. This is pretty similar to what happens with CISC instructions, except with much more complicated instructions. However, a RISC would be able to have these operations implemented in the hardware rather than needing microcode controllers, saving time. With all instructions happening at one cycle each, it is easier to look ahead and see where the program will be in a few clock cycles.

↳ **Register to Register Operations**

↳With RISC, typically only the LOAD and STORE instructions will access memory. This means that with RISC, there would only be the ability to ADD two registers together, rather than having the ability to have ADD pull in operands from memory, and the same for all computation operations. This greatly simplifies the control unit, and the instruction set itself. Furthermore, this encourages optimization of register use, making sure all computation operands are stored in high speed registers. Due to programs having relatively low amounts of variables at play at any given time, most operands can be kept in registers for long periods of time.

↳ **Simple Addressing Modes**

↳ Most RISC computers have only a few addressing modes, with the rest of the more complicated addresses needing to be calculated using assembly operations when needed. This is because by far the majority of addressing modes actually used are the simple ones, such as immediate and direct.

↳ **Simple Instruction Formats**

↳ With RISC only a couple instruction formats are used. Word size is generally fixed, (as opposed to having unary, binary, ternary etc instructions), and special locations, such as the opcode, the addressing fields, or the register signifiers are in fixed locations. This can greatly simplify the stages of the instruction cycle, optimize the fetching of each operation, and can allow for multiple steps to happen simultaneously.

↳ **Real world Examples**

↳Theoretically more effective compilers can be developed, with less wasteful and expensive
operations, and rather many sequences of assembly commands well suited for fast register
operations.

↳RISC processors are more responsive to interrupts (when the computer is called to do
something else in the middle of processing something) because all the instructions are 1 cycle
length and do simple operations. This is much harder to do on a CISC computer.

↳ CISC

↳Characteristics

↳complex instruction sets were used throughout the years, studies began to be done to find out whether these types of designs actually accomplished these things.

- **Operations performed:** Which operations are actually being used?
    - ASSIGN 15%
    - • LOOP 26%
    - • CALL 45%
    - • IF 13%
    - • OTHER 1%
        - Basically, these numbers are saying that most programs consist of mainly a bunch of function calls, loops, ifs, and assignment statements. If you look at many of the programs that you have written, you will find that this is probably pretty accurate. The big complex operations that CISC is designed to be efficient for? These are actually used about 1% of the time. If we can instead ensure that these core operations are extremely optimized, we could probably still get most of any performance gains. Furthermore, despite many CISC operations being designed for a massive number of arguments, it was found that most calls had less than 6 arguments.

- **Operands**: Type and frequency of operands used as well as which addressing modes are used the most. (Remember these CISCs have dozens of addressing modes and several variable length instructions)
    - Integer Constant 20%
    - Primitive Variable 55%
    - Array/Structure 25%
        - These numbers showed that the majority of data references are to simple variables, furthermore, 80% of these variables were local to the procedure they were used in.This indicates that a prime candidate for optimization is the mechanism for storing and accessing local variables, and that instructions for handling complex data types are probably not as important as CISC designers thought necessary.
- **Execution Sequencing**: Control statements (if, function calls) affecting the ability to parallel process.
    - The instruction set, or basic set of operations that the computer could perform, was fairly simple. Any advanced operation would require the execution of several instructions. Due to the instruction cycle having a fairly fixed number of steps and time (other than execution) this caused complex operations to take much longer than simple ones.
    - As high-level languages with each line requiring more complex operations became widespread, an issue began to form. This issue, known as the semantic gap, was that there was a difference between the most common operations in the primary programming languages being used and the instruction set of the computer. Each programming line took many instructions, and programs began to slow down.
    - This gap produces symptoms such as:
        - • execution inefficiency
        - • excessive machine program size

        - • compiler complexity

- **Real life examples**
    - Intel. Intel is the only chipmaker that uses CISC and continues to do so to support backwards compatibility with older chipsets. It makes enough money to where they can say "too bad" to all of the RISC benefits
    - These studies concluded that the attempt to make the instruction set architecture close to High Level Languages is not the most effective design strategy. Rather the HLLs can be best supported by optimizing the performance of the most time-consuming features of typical HLL programs.
        - 1. We need a lot of general-purpose registers, which allow us to hold all our local variables rather than accessing the memory or cache.
        - 2. There are a lot of calls and branches, which means that looking ahead at the next several instructions to precompute things will not be sufficient.
        - 3. A reduced instruction set is better suited for computer performance. You should see why from the ensuing discussion.
        - cuda
- **Identify concrete real-world examples of RISC and CISC architectures**
    - Intel. Intel is the only chipmaker that uses CISC and continues to do so to support backwards compatibility with older chipsets. It makes enough money to where they can say "too bad" to all of the RISC benefits
- **Connect the concepts of pipelining and instruction sets**
    - Pipelining
        - Pipelining, a standard feature in RISC processors, is much like an assembly line. Because the processor works on different steps of the instruction at the same time, more instructions can be executed in a shorter period of time.
        - RISC Pipelines
            - fetch instructions from memory
            - read registers and decode the instruction
            - execute the instruction or calculate an address
            - access an operand in data memory
            - write the result into a register
        - Thus, the length of the pipeline is dependent on the length of the longest step. Because RISC instructions are simpler than those used in pre-RISC processors (now called CISC, or Complex Instruction Set Computer), they are more conducive to pipelining. While CISC instructions vary in length, RISC instructions are all the same length and can be fetched in a single operation. Ideally, each of the stages in a RISC processor pipeline should take 1 clock cycle so that the processor finishes an instruction each clock cycle and averages one cycle per instruction (CPI).

- **Identify the role of "planning for parallelism" in modern instruction set design**
  - **SISO**
    - Single instruction single data stream
      - 1 instruction set
      - 1 data stream
    - Uniprocessor
      - 1 core processor
  - **SIMD**
    - Single instruction multiple data
      - 1 instruction set
        - Meaning there is 1 program with a single program counter we go through
      - >1 data stream operated on
    - Most common uses
      - Most common example is a vector processor
        - Executing an add instruction but it operates on 2 sources and one destination vector of things
      - Modern processors also,have thus multimedia extension instruction
        - MMX or SSE
  - **MISD**
    - Multiple instruction streams one 1 data stream
      - Not really used much
    - Know as a stream processor
  - **MIMD**
    - More than one instruction set and more than one data stream
      - Known as a multiprocessor
        - Most processors bought today
- **Contrast the philosophies behind RISC and CISC, and the tradeoffs of each**
  - There are several problems with comparing RISC to CISC:
    - • There are no pairs of RISCs and CISC computers of comparable cost, level of technology, gate complexity, compiler, operating system support, and so on.
    - • No definitive test set of programs exist, performance varies with the program.
    - • It is difficult to sort out hardware effects from effects due to skill of compiler writing.
    - • Most RISC analysis has been done on academic "toy" machines rather than commercial products. Most commercial products advertised as RISC are actually a mix of RISC and CISC.
  - Recently the argument between the two camps has died down. Over time, the two technologies have basically converged. As we have been able to fit more stuff on a chip (due to Moore's law) RISC systems have incorporated some of the most useful "complex" operations, CISC designers have added many more

general-purpose registers and have had more of an emphasis on instruction pipeline design.

- **Identify how the best features of RISC and CISC architectures have converged in more modern versions of architectures (x86, ARM, etc) of the 21st century**

Week 10-11: CPU Parallelization

Learning Objectives (Dr. Feister edits):

- **Compare and contrast multiple levels of hardware parallelization**
    - **Multithreaded native applications (thread-level parallelism**)
        - Multithreaded applications are characterized by having a small number of highly threaded processes.
    - **Multiprocess applications (process-level parallelism):**
        - Multiprocess applications are characterized by the presence of many single-threaded processes.
    - **Java applications:**
        - Java applications embrace threading in a fundamental way. Not only does the Java language greatly facilitate multithreaded applications, but the Java Virtual Machine is a multithreaded process that provides scheduling and memory management for Java applications.
    - **Multi-instance applications (application-level parallelism)**
        - Even if an individual application does not scale to take advantage of a large number of threads, it is still possible to gain from multicore architecture by running multiple instances of the application in parallel. If multiple application instances require some degree of isolation, virtualization technology (for the hardware of the operating system) can be used to provide each of them with its own separate and secure domain
    - Hardware level works upon **dynamic parallelism**, whereas the software level works on **static parallelism**.
    - **Dynamic Parallelism**
        - Dynamic parallelism means the processor decides at run time which instructions to execute in parallel
    - **Static parallelism**
        - whereas static parallelism means the compiler decides which instructions to execute in parallel.
- **Justify why multicore CPUs were not heavily developed prior to 2000, and yet are everywhere today**
    - From about 1970-2000 any code we wrote would essentially run twice as fast every two years. As programmers we had to do basically nothing as long as our

code could run on the new systems to get these performance benefits. However, the time for single core computers to keep doubling in speed like this has come and gone.
    - Imagine you wrote a single threaded, computationally advanced program in 2000. Let's take a look it how it would have performed on high end, consumer CPUs throughout the years.
    - **High End Intel CPUs Through the Years:**
        - 2000: Pentium 4: 1.5 Ghz, Single Core
        - 2002: Pentium 4: 2.4 Ghz, Single Core
        - 2005: Pentium Extreme: 3.73 Ghz, Single Core
        - 2010: i5: 2.5 Ghz 4 Cores
        - 2014: i7: 3.5 Ghz 6 Cores
        - 2019: i9: 3.7 Ghz 8 Cores
    - **Performance on Single Core:**
        - 2000: Pentium 4: 1.5 Ghz, Single Core Baseline
        - 2002: Pentium 4: 2.4 Ghz, Single Core 1.6x
        - 2005: Pentium Extreme: 3.73 Ghz, Single Core 2.5x
        - 2010: i5: 2.5 Ghz 4 Cores 1.7x
        - 2014: i7: 3.5 Ghz 6 Cores 2.2x
        - 2019: i9: 3.7 Ghz 8 Cores 2.5x
    - 20 years later and your program runs 2.5x faster. It also ran this fast in 2005, 15 years ago.
- **Apply your knowledge of the control unit to instruction pipelining**
    - A multicore processor, also known as a chip multiprocessor, combines two or more processor units (called cores) on a single piece of silicon (called a die). Typically, each core consists of all of the components of an independent processor, such as registers, ALU, pipeline hardware, and control unit, plus L1 instruction and data caches. In addition to the multiple cores, contemporary multicore chips also include L2 cache and increasingly, L3 cache. The most highly integrated multicore processors, known as systems on chip (SoCs), also include memory and peripheral controllers.
- **Appraise challenges of programming for multicore systems, especially the two challenges of parallel access to shared memory and of achieving nominal performance gains**
    - Increase in Parallelism and Complexity The organizational changes in processor design have primarily been focused on exploiting ILP, so that more work is done in each clock cycle. These changes include, in chronological order (Figure 18.1):
        - **Pipelining**: Individual instructions are executed through a pipeline of stages so that while one instruction is executing in one stage of the pipeline, another instruction is executing in another stage of the pipeline. Superscalar: Multiple pipelines are constructed by replicating execution resources. This enables parallel execution of instructions in parallel pipelines, so long as hazards are avoided.
            - Design
                - The same point of diminishing returns is reached with SMT, as the complexity of managing multiple threads over a set

of pipelines limits the number of threads and number of pipelines that can be effectively utilized.
        - Similar to the people in the fields constantly working you have so many people how do you start them off working together to get a task done without any people on the side not doing anything
    - Nominal Performance gains
        - his same point of diminishing returns is reached with SMT, as the complexity of managing multiple threads over a set of pipelines limits the number of threads and number of pipelines that can be effectively utilized. SMT's advantage lies in the fact that two (or more) program streams can be searched for available ILP.
        - There is a related set of problems dealing with the design and fabrication of the computer chip. The increase in complexity to deal with all of the logical issues related to very long pipelines, multiple superscalar pipelines, and multiple SMT register banks means that increasing amounts of the chip area are occupied with coordinating and signal transfer logic. This increases the difficulty of designing, fabricating, and debugging the chips. The increasingly difficult engineering challenge related to processor logic is one of the reasons that an increasing fraction of the processor chip is devoted to the simpler memory logic.
    - physical memory is partitioned between CPU and GPU. If an application thread is running on a CPU that demands GPU processing, the CPU explicitly copies the data to the GPU memory. The GPU completes the computation and then copies the result back to CPU memory.
        - Issues of cache coherence across CPU and GPU memory caches do not arise because the memory is partitioned. On the other hand, the physical handling of data back and forth results in a performance penalty.
- **Distinguish processes from threads**
    - **Process**: An instance of a program running on a computer. A process embodies two key characteristics:
        - **Resource ownership:** A process includes a virtual address space to hold the process image; the process image is the collection of program, data, stack, and attributes that define the process. From time to time, a process may be allocated control or ownership of resources, such as main memory, I/O channels, I/O devices, and files.
        - **Scheduling/execution:** The execution of a process follows an execution path (trace) through one or more programs. This execution may be interleaved with that of other processes. Thus, a process has an execution state (Run ning, Ready, etc.) and a dispatching priority and is the entity that is scheduled and dispatched by the operating system.
            - **Process switch**: An operation that switches the processor from one process to another, by saving all the process control data,

registers, and other information for the first and replacing them with the process information for the second.

- **Thread**: A dispatchable unit of work within a process. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching). A thread executes sequentially and is interruptible so that the processor can turn to another thread.
  - **Thread switch**: The act of switching processor control from one thread to another within the same process. Typically, this type of switch is much less costly than a process switch.
- Thus, **a thread is concerned with scheduling and execution, whereas a process is concerned with both scheduling/execution and resource ownership.** The multiple threads within a process share the same resources. This is why a thread switch is much less time consuming than a process switch. Traditional operating systems, such as earlier versions of unix, did not support threads. Most modern operating systems, such as Linux, other versions of unix, and Windows, do support thread. A distinction is made between user-level threads, which are visible to the application program, and kernel-level threads, which are visible only to the operating system. Both of these may be referred to as explicit threads, defined in software.

Week 12: Massively Parallel Processors

Learning Objectives (Dr. Feister edits):

- **Summarize differences between CPUs, GPUs, and NPUs**
  - **CPU**
    - Central processing unit
    - Supports parallelism
      - Designed to do general computing tasks at hand(doing the basic day to day functions) with varying degrees of speed
    - Few but large cores
  - **NPU**
    - Neural Processing Units (NPUs)
      - Tensor processing unit, neural network processing unit, or intelligent processing units
    - Designed to learned with data on how to solve a problem
      - Without being programmed
      - Cuda cores
    - Practical use
      - Phones use this
        - Pattern recognition
        - Image processing
        - Apple Samsung huawei and qualcomm use this

    - **GPU**
        - Graphics processing unit
        - Thousands of smaller cores
        - A GPU is a vector machine. You can give it a long list of data — a 1D vector — and run computations on the entire list at the same time. This way, we can perform more computations per second, but we have to perform the same computation on a vector of data in parallel. This kind of computation lends itself well to graphics or crypto-mining, where one job must be performed many times.
    - **Similarities**
        - Both silicon based micro processors mounted to a PCs with heat sinks attached to them
    - **Differences**
        - CPUs are good at multitasking quickly
            - Modern PCs are general purpose that can do the basics
                - Due to this itt needs to be able to do a lot of things simultaneously
        - GPUs are there for only maintaining or rendering graphics
            - This means that they do millions of calculations at the same time in parallel
                - They focus on raw throughput than the cpu which is designed to be much more versatile and balanced all the different work loads
        - Architecture wise GPUs feature lots and lots of identical compute units that deal with solving similar math problems
            - This lead to things like gpgpus (general purpose computing on graphical processing units) being created
                - Bitcoin mining and Stanford universities at home project is an example of this
            - You can't run an operating system in a graphics card because the uses are very different
- **Define massive parallelism, and contrast with multicore CPU parallelism**
    - massive parallelism
        - hardware designed such that thousands or millions of processes can run at the same time.
        -
    - multicore CPU parallelism
        - One flavor of massively parallel hardware is known as the GPU, or graphics processing unit. You'll see why graphics requires massive parallelism in this week's readings, and also how the GPU hardware has been evolving towards running general-purpose software where parallelism can provide big speedups.
        -
- **Identify real-world trends in heterogeneous computer processor organizations**

    - heterogeneous multicore design is the use of both CPUs and graphics processing units (GPUs) on the same chip. GPUs are discussed in detail in the following chapter. Briefly, GPUs are characterized by the ability to support thousands of parallel execution threads. Thus, GPUs are well matched to applications that process large amounts
    - **such as CUDA** (Compute Unified Device Architecture), these new processors are increasingly being applied to improve the performance of general-purpose and scientific applications th**at involve large numbers of repetitive operations on str**uctured data.
    -

- **Assess why certain problems benefit from massive parallelism and others do not**
    -
- **List several real-world applications particularly well suited for each of CPUs, GPUs, and NPUs**
    - **CPU**
        - Used in any basic activity
            - Looking things up online
            - Using Spreadsheets
    - **GPU**
        - Best known for video games, video rendering, graphics
        - accelerate video, digital image, and audio signal processing, statistical physics, scientific computing, medical imaging, computer vision, neural networks and deep learning, cryptography, and even intrusion detection
            - Fortnite
            - Minecraft
            - Just dance
    - **NPU**
        - This is used in smartphones
- **Examine tradeoffs of developing specific hardware for specific problems, vs. general hardware for general problems**
    - One of the main problems would be using the technology interchangeably. Because a GPU can't do a simple task of multi-function, and instead only focuses on the one task it can manage this one task extremely well and is angled to do it millions of times fast. The downside is that the items of tech are going to be bigger in size to accommodate each of the specific tasks.

Week 13: Quantum Computing

Learning Objectives (Dr. Feister edits):

- **Assess the current state of real-world quantum computing hardware**
- **Evaluate future possibilities for quantum computing**
- **Describe how quantum computing is different from classical computing**

How do you program an atom on quantum computers?

Week 14: Exotic Orgs & Archs

Learning Objectives (Dr. Feister edits):

- **Explore and evaluate radical ideas for computer organization**
- **Identify several non-electronic methods for computing**
- **Compare and contrast these ideas with conventional computer organization**