CISC versus RISC
Architecture
Abstract

As modern RISC and CISC systems seek ways to increase performance, they are more often embracing common ideas, such as super scalar pipelining, small instruction format, and more dependence on compilers. These systems which once embodied opposite design principles are becoming more similar. CISC designs, while historically placing as much complexity as possible onto the hardware to minimize software size and compiler dependence, have now embraced RISC-like methodologies. RISC systems, which developed in response to increasingly complex CISC systems by focusing on the minimization of hardware and utilization of a uniform instruction set and robust compiler, have replicated internal components and added complexities whose inclusion was traditionally favored by CISC designers. This paper compares these two different architectures in a comprehensive way and try to capture the purely architectures advantages or RISC and CISC and the development trends of the future architectures.
# Table of Contents

1.0 Introduction ................................................................................. 3

2.0 History of CISC and RISC Architectures .............................. 4
  2.1 Storage and memory ............................................................ 5
  2.2 Compiler .............................................................................. 5
  2.3 VLSI .................................................................................. 6

3.0 CISC Architecture ................................................................. 6
  3.1 Complex Instruction Example .............................................. 7
  3.2 Complex Instruction Example .............................................. 9

4.0 RISC Architecture ................................................................... 11
  4.1 Pipelining in Depth ............................................................. 13
    4.1.1 Latency and throughput .................................................. 13
    4.1.2 Bubbles ....................................................................... 14
    4.1.3 Branching in pipelined processors ................................. 15
    4.1.4 Branch Delay Slots ....................................................... 16
  4.2 Reduced Instruction Example ............................................... 17

5.0 Differences Between CISC and RISC Architectures ............ 18

6.0 Similarities Between CISC and RISC Architectures ............ 20

7.0 Conclusion .............................................................................. 20
1.0 Introduction

RISC and CISC stand for two different competing philosophies in designing modern computer architecture. The debate between them has been going on for a long time and will likely to continue. In order to understand the historical and technological context out of which RISC and CISC developed, it is first necessary to understand the state of the art in VLSI, storage, and compilers in the late 70’s and early 80’s. These three technologies defined the technological environment in which researchers worked to build the fastest machines.

The primary goal of developing computer processors is performance. Every microprocessor manufacturer wants to have the fastest offerings. There is one standard formula, which describes how performance is measured for these systems, in terms of the amount of work done by the processor in a given amount of time:

\[
\text{time}_{\text{program}} = \text{time}_{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}
\]

The two design philosophies analyzed in this research paper traditionally focus on one of the terms in this equation to improve performance. There are tradeoffs to any processor modification, however the question is what these tradeoffs are and how much they affect the overall system performance. The cycles per instruction is determined by a combination of the instruction set, hardware organization such as the degree of pipelining and the structure of the cache-memory system and the compiler. The number of instructions is a function of the compiler and the target architecture. The cycle speed is a function of the underlying technology used to build the processor.

The RISC advocates tries to reduce the time spent on a program by reducing the cycles per instruction and allowing for a slight increase in the number of instructions in a program. The CISC advocates take the opposite approach, by trying to reduce the number of instructions per program. The heart of the RISC vs. CISC debate is which philosophy produces the better processor.
2.0 History of CISC and RISC Architectures

The IBM 360 system, created in 1964, was probably the first modern processor system, which initiated the idea of computer architecture in computer science and adopted microcoded control. Microcoded control facilitated the use of complex instruction sets and provided flexibility. Complex Instruction Set Computer (CISC) was primarily motivated by a desire to reduce the semantic gap between the machine language of the processor and the high level languages. The theory was that such a processor would have to execute fewer instructions therefore would have better performance.

CISC computers are based on complex instruction set in which instructions are executed by microcode. Microcode allows developer to change hardware designs and still maintain backward compatibility with instructions for earlier computers by changing only the microcode, thus make a complex instruction set possible and flexible. Although CISC designs allow a lot of hardware flexibility, the supporting of microcode slows microprocessor performance.

However, in the middle of 1970’s, people began to doubt the design philosophy behind CISC. With more and more complex instructions sets, decoding and execution of such instructions were complicated and time consuming, also, the expensive overhead brought by them slowed down the execution of those more frequently used simple instructions. Moreover, with the development of high-level languages, making good use of the instruction set posed a problem to compilers. On the other hand with the declining cost of memory devices and improved compiler technology, it maybe feasible to consider simplifying the instruction set with the cost of larger code size and higher memory bandwidth requirements. Based on this observation, RISC (Reduced Instruction Set Computer) chips evolved as a reaction at CISC chips.

The design philosophies behind RISC chip is that 20 percent of the computer's instructions do about 80 percent of the work. In a CISC chip, many very complex instructions never or seldom used, but they make the control unit extremely complex and therefore have a high control unit delay. A RISC instruction set
includes fewer and simpler instructions with hard-wired control, simpler processor pipeline, a larger number of registers, a smaller transistor count which makes it easier to design and cheaper to produce, and higher clock rate. However, over time, the battle over RISC and CISC became blur, though pure RISC machine may outperform pure CISC machine, but both of each have some bad characteristics which interfere their further improvement of performance. Recently, the trend is migrating toward each other; RISC machines may adopt some traits from CISC, while CISC may also adopt some traits from RISC.

2.1 Storage and memory

It’s hard to underestimate the effects that the state of storage technology had on computer design in the 70’s and 80’s. In the 1970’s, computers used magnetic core memory to store program code; core memory was not only expensive, it was agonizingly slow. After the introduction of RAM things got a bit better on the speed front, but this didn’t address the cost part of the equation. Consider the fact that in 1977, 1MB of DRAM cost about $5,000. By 1994, that price had dropped to under $6. In addition to the high price of RAM, secondary storage was expensive and slow, so paging large volumes of code into RAM from the secondary store impeded performance in a major way. The high cost of main memory and the slowness of secondary storage conspired to make code a serious issue. Good code was compact code; you needed to be able to fit all of it in a small amount of memory. Because RAM counted for a significant portion of the overall cost of a system, a reduction in code-size translated directly in to a reduction in the total system cost. In the early 90’s, RAM accounted for around %36 of the total system cost, and this was after RAM had become quite a bit cheaper.

2.2 Compiler

The compiler’s job was fairly simple at that point: Translate statements written in a high level language, like C or PASCAL, into assembly language. The assembly language was then converted into machine code by an assembler. The
compilation stage took a long time, and the output was hardly optimal. As long as the high level language assembly translation was correct.

2.3 VLSI
The state of the art in Very Large Scale Integration (VLSI) yielded transistor densities that were low by today’s standards. It was not possible to fit too much functionality onto one chip. Back in 1981 a million transistors on a single chip was a considered very high. Because of the scarcity of available transistor resources, the CISC machines like the VAX, had their various functional units split up across multiple chips. This was a problem, because the delay-power penalty on data transfers between chips limited performance. A single-chip implementation would have been ideal, but it wasn’t feasible without a radical rethinking of current designs.

3.0 CISC Architecture
An important factor in computer design prior to 1980 was that all memory, including the memory to store program instructions, was very expensive. As hardware was getting cheaper, the software costs were spiraling out of control. Researchers decided that one way to stave off impending doom was to shift the burden of complexity from the increasingly expensive software level to the increasingly inexpensive hardware level. That is to implement common functions or operations in hardware to minimize the number of steps that a programmer must write every time to use the function. After all, hardware was cheaper than the programmer time. This idea of moving complexity from the software realm to the hardware realm is the driving idea behind CISC, and almost everything that a true CISC machine does is aimed at this end.

Some researchers suggested that the way to make programmers and compiler-writers jobs easier was to close the semantic gap between statements in a high-level language and the corresponding statements in assembler. Closing the semantic gap was a fancy way of saying that system designers should make assembly code look more like C or PASCAL code.
The discussion so far has focused more on the economic advantages of CISC, while ignoring the performance side of the debate. The CISC approach to increase performance is rooted in the philosophy to move complexities from software level to the hardware level.

Consider the performance equation:

\[
\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}
\]

The above equation is a commonly used metric for gauging a computer’s performance. Increasing performance means reducing the term on the left side of the equation, since the less time it takes to run a program, the better the machine’s performance. A CISC machine tries to reduce the amount of time it takes to run a program by reducing the number of instructions per program. Researchers thought that by reducing the overall number of instructions that the machine executes to perform a task you could reduce the overall amount of time it takes to finish that task, and thus increase performance.

So decreasing the size of programs not only saved memory, it also saved time because there were fewer lines of code to execute. While this approach proved fruitful, it had its drawbacks.

### 3.1 Complex Instruction Example

Figure 1 depicts a block diagram that represents the storage scheme for a generic computer. The main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4. The execution unit is responsible for carrying out all computations. However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F). To find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 and then store the product back in the location 2:3.
The code required to multiply the two numbers spells out explicitly the steps that assembler has to take to multiply the contents of the two memory locations together. It tells the computer to load the two registers with the contents of main memory, multiply the two numbers, and store the result back into main memory.

1. MOVE [A, 2:3]  
2. MOVE [B, 5:2]  
3. MUL [A, B]  
4. MOVE [2:3, A]  

The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a specific instruction and let this specific instruction be called "MULT". When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction:

\[
\text{MULT 2:3, 5:2}
\]
MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher level language. For instance, let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a * b."

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is on building complex instructions directly into the hardware.

Changing from four instructions to one is a big savings. However, the system still has to load the contents of the two memory locations into registers, multiply them, and write them back out—there's no getting around all that—but all of those lower-level operations are done in hardware and are invisible to the programmer. So all that complicated work of shuffling memory and register contents around is hidden; the computer takes care of it behind the scenes. This is the primary objective of the CISC philosophy that is moving functionality from software level into the hardware level.

### 3.2 Complex Instruction Example

To better understand the motivations behind increasing the complexity of a machine's instructions, consider the following example to take the cube of 20 and store it in a variable.

- **MOVE [destination register, integer or source register].**
  This instruction takes a value, either an integer or the content of another register, and places it in the destination register. So MOVE [D, 5] would place the number 5 in register D. MOVE [D, E] would take whatever number is stored in E and place it in D.

- **MUL [destination register, integer or multiplicand register].**
  This instruction takes the content of the destination register and multiplies it by either an integer or the content of multiplicand register, and places
the result in the destination register. So MUL [D, 70] would multiply the content of D by 70 and place the results in D. MUL [D, E] would multiply the contents of D by the content of E, and place the result in D.

<table>
<thead>
<tr>
<th>Statements in High Level Language</th>
<th>Statements in Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A = 20</td>
<td>1. MOVE [A, 20]</td>
</tr>
<tr>
<td>2. B = CUBE (A)</td>
<td>2. MUL [A, A]</td>
</tr>
<tr>
<td></td>
<td>3. MUL [A, A]</td>
</tr>
<tr>
<td></td>
<td>4. MOVE [B, A]</td>
</tr>
</tbody>
</table>

This example actually finds $20^4$. The above example takes four statements in assembly to do the work of two statements in High Level Language. The four statements are required since the assembly computer has no instruction to take the CUBE of a number. That is it requires additional instructions to find the cube of A. Then, the following observation can be made:

- If the High Level Language program uses the CUBE function extensively, then when compiled the assembler program will be quite a bit larger than the High Level Language program. Since the assembly computer doesn’t have too much memory, this can turn to be a major problem.

- In addition, it will take the compiler a long time to translate all the CUBE statements into MUL instructions.

- Finally, if a programmer decides to forget about High Level Language and just write code in assembler, the programmer will have more typing to do, and the fact that the code is so long makes it harder to debug.

One way to solve this problem would be to include a CUBE instruction in the next generation of the assembler architecture. That is the assembler has an instruction that looks as follows:

- CUBE [destination register, multiplicand register].
This instruction takes the contents of the multiplicand register and cubes it. It then places the result in the destination register. So CUBE [D, E] takes whatever value is in E, cubes it, and places the result in D.

<table>
<thead>
<tr>
<th>Statements in High Level Language</th>
<th>Statements in Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A = 20</td>
<td>1. MOVE [A, 20]</td>
</tr>
<tr>
<td>2. B = CUBE (A)</td>
<td>2. CUBE[B, A]</td>
</tr>
</tbody>
</table>

So now there is a one-to-one correspondence between the statements in High Level Language, and the statements in assembly. The semantic gap has been closed, and the compiled code is smaller, easier to generate, easier to store, and easier to debug. However, the assembly computer still cubes numbers by multiplying them together repeatedly in hardware, but the programmer doesn’t need to know that. All what the programmer needs to know is that there is an instruction on the machine that can cube a number. This is the fundamental CISC tenet of moving complexity from the software level to the hardware level.

4.0 RISC Architecture

As the CISC architecture get more complex, more chips were needed to support the CISC architecture. Designers were under pressure to simplify their designs by using only one chip for the CPU. But, since chips at early age had limited amount of transistors, scientists began analyzing instruction streams to determine which instructions were more frequently executed and which instructions were less executed. Based on that analysis, it was found that the greatest amount of time was spent executing simple instructions, doing loads and stores. In addition, the compiler rarely used the complex instructions that CISC used. This study led to optimizing the instructions that were executed the most. However, since software prices were on the decline, it became unnecessary for the processor to do most of the tasks. This was the beginning of the RISC architecture. In addition to optimizing common instructions, designers also reduced the amount of clock cycles used to only one clock cycle per instruction.
Pipelining is the practice of running multiple instructions during one clock cycle. Pipelining is only effective if the instructions do not vary in complexity. In order for the instructions to be pipelined they have to be able start and finish at the same times as each other. If done correctly, this can greatly increase performance. In the performance equation the RISC architecture tries to reduce the number of clock cycles while having more instructions. This turn out to be much faster than the CISC architecture’s method.

With RISC computers having equal instruction lengths, this fact was used to full advantage through pipelining. Executing an instruction is not a single step. Instead, execution takes 5 steps:

- The instruction is fetched from memory
- The instruction is recognized
- The processor fetches operand the instruction needs
- The instruction is actually executed
- The processor writes back the result to wherever it is supposed to go.

Pipelining helps the process along by putting instructions back to back, such that as the first instruction is being written back, the second is being executed; the third is having its operands fetched, and so forth. This is much quicker than letting a single instruction going through all 5 stages before the next instruction is stepped.

Another way to look at pipelining is to view pipelining like those little tunnels at playground. Now assume a bunch of children want to see how fast they can all go through the tunnels. If there is only one tunnel, each child has to wait for the child before to be partially through. However, if there were five tunnels, the children could split up and each goes through different tunnels to finish much more quickly. Such is the idea of computer pipelining. However the order in which instructions are executed does matter. By doing all this splitting for pipelining, instructions can get separated from other instructions that they should be with. This is one of the problems of pipelining. Taking into account this problem, the computer has to figure out how to split instructions up to be
pipelined without B coming before A, and so forth. While such considerations do make pipelining more difficult, it is still worth the effort for the speed it generates.

4.1 Pipelining in Depth

A pipelined processor doesn't wait until the result from a previous operation has been written back into the register files or main memory. It fetches and starts to execute the next instruction as soon as it has fetched the first one and dispatched it to the instruction register. When the processor is performing an add instruction, there is no need for the processor to wait for the add operation to complete before it starts fetching the next instruction. So a pipelined processor will start fetching the next instruction from memory as soon as it has latched the current instruction in the instruction register.

<table>
<thead>
<tr>
<th>IF</th>
<th>Instruction Fetch</th>
<th>Time</th>
<th>IF</th>
<th>DEC</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch the instruction from memory</td>
<td>1</td>
<td>I1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>I2</td>
<td>I1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>Decode and Operand Fetch</td>
<td>3</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Decode it and fetch operands from the register file</td>
<td>4</td>
<td>I4</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
</tr>
<tr>
<td></td>
<td>Execute</td>
<td>5</td>
<td>I5</td>
<td>I4</td>
<td>I3</td>
<td>I2</td>
</tr>
<tr>
<td></td>
<td>Execute the instruction in the ALU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>WriteBack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write the result back in to a register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Population of the pipeline at each clock cycle: \(i_1, i_2, \ldots\) are successive instructions in the instruction stream.

4.1.1 Latency and throughput

Each instruction will take \(n\) cycles to complete. However, the throughput has been increased to an instruction every cycle from one every \(n\) cycles. The latency or lag between the times an instruction is given to the processor and the time that it emerges with a result has not changed. Therefore, adding a degree of parallelism to the system increases throughput.
4.1.2 Bubbles

<table>
<thead>
<tr>
<th>TIME</th>
<th>IF</th>
<th>DEC</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I2</td>
<td>I1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>I4</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
</tr>
<tr>
<td>5</td>
<td>I4</td>
<td>I3</td>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>I4</td>
<td>I3</td>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>I5</td>
<td>I4</td>
<td>I3</td>
<td>I2</td>
</tr>
</tbody>
</table>

High throughput can be achieved if every instruction takes exactly one cycle to execute. Although most arithmetic instructions in a RISC machine complete in a single cycle, some complex instructions, such as divide, take on more than one cycle to complete. Such an instruction remains in execute stage for more than one cycle and creates a pipeline bubble ahead of it in the pipeline. In this case the divide instruction $i_2$ is a long-latency instruction that is taking 3 cycles in the execute stage. A bubble develops ahead of it in the pipeline as instructions ahead of it move on to the next stage.

Pipeline bubbles are caused by:

- Long latency instructions, such as division, floating point operations and math functions, which take more than one cycle in the execution stage.
- Memory reads, which take many cycles in the Decode stage.
- Memory writes, which may take extra cycles in the WriteBack stage, and Branches.
4.1.3 Branching in pipelined processors

<table>
<thead>
<tr>
<th>TIME</th>
<th>IF</th>
<th>DEC</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I2</td>
<td>I1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>I4</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
</tr>
<tr>
<td>5</td>
<td>i_a</td>
<td></td>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>i_b</td>
<td>i_a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Efficient handling of branch instructions presents a significant challenge to a computer architect. The simplest branch is an unconditional branch, such as the return from a procedure or function, requires a number of instructions in the pipeline to be squashed.

When \( i_2 \) the execution stage and the new PC are being calculated, instructions \( i_3 \) and \( i_4 \) have already entered the pipeline and must be squashed. This creates a series of bubbles in the pipeline until the target of the branch instruction, \( i_a \), and its successors can be fetched from memory.

The diagram shows the cost of branches fairly dramatically - a large number of bubbles are present for quite a few cycles. Although this example makes an optimistic assumption that the new instruction stream could be accessed within 3 cycles, this assumption is not always practical since if the new instructions have to come from memory rather than cache.
4.1.4 Branch Delay Slots
Since branches occur very frequently in most programs, probably on the order of one every 10 instructions, an instruction is added following the branch instruction. This instruction is called a branch delay slot. It was soon realized that, since the instruction in this slot has progressed well down the pipeline anyway, if it was guaranteed that it would be executed, some improvement in performance would result.

```
mul  $2, $2, $4
add  $3, $1, $4
ret
or    $1, $1, $1 Branch Delay Slot
sub $4, $1, $6
```

Compiler reorders instructions

```
mul  $2, $2, $4
ret
add  $3, $1, $4 Always Executed
sub $4, $1, $6
```

To improve the performance, the compiler is asked to move an instruction that must be executed which precedes the branch into the branch delay slot where it will be executed while the branch target is being fetched.

This example reflects the modern trend in computer architecture. The objective is to expose more details of the underlying machine to the compiler and to let the compiler generate the most efficient code. Recently, compilers are rapidly becoming more capable and optimizing compilers routinely performs much more complex instruction reordering operations.
4.2 Reduced Instruction Example

To see the different between the CISC and RISC approach, this example find the product of two numbers one stored in location 2:3 and the other stored in location 5:2 and then store the product back in the location 2:3.

<table>
<thead>
<tr>
<th>1:1</th>
<th>1:2</th>
<th>1:3</th>
<th>1:4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:1</td>
<td>2:2</td>
<td>2:3</td>
<td>2:4</td>
</tr>
<tr>
<td>3:1</td>
<td>3:2</td>
<td>3:3</td>
<td>3:4</td>
</tr>
<tr>
<td>4:1</td>
<td>4:2</td>
<td>4:3</td>
<td>4:4</td>
</tr>
<tr>
<td>5:1</td>
<td>5:2</td>
<td>5:3</td>
<td>5:4</td>
</tr>
<tr>
<td>6:1</td>
<td>6:2</td>
<td>6:3</td>
<td>6:4</td>
</tr>
</tbody>
</table>

RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the MULT command used in the CISC approach could be divided into three separate commands:

- **LOAD**, which moves data from the memory to a register.
- **PROD**, which finds the product of two operands located within the registers.
- **STORE**, which moves data from a register to the memory.

In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

1. LOAD A, 2:3
2. LOAD B, 5:2
3. PROD A, B
4. STORE 2:3, A

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the
assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form. However, the RISC strategy also brings some very important advantages. Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle MULT command. These RISC reduced instructions require less transistors of hardware space than the complex instructions, leaving more room for general-purpose registers. Because all of the instructions execute in a uniform amount of time that is one clock, pipelining is possible. Separating the LOAD and STORE instructions actually reduces the amount of work that the computer must perform. After a CISC-style MULT command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

5.0 Differences between CISC and RISC Architectures

CISC designers shifted the burden of processing to the hardware. To achieve increased performance, tradeoffs were made in favor of smaller code at the expense of the clock cycles per instruction. Software generated for these systems is typically smaller than that for RISC counterparts, because the processor has more functionality built-in. A more complicated processor, however, means more transistors and more time to complete an instruction. The main features of CISC architecture are the following:

- Emphasis on hardware.
- Any instruction may reference memory.
- Many instructions and addressing modes.
- Variable instruction formats.
- Single register set.
- Multi clock cycle instructions.
- Micro-program interprets instructions.
- Complexity is in the micro-program.
- Less to no pipelining.
- Program code size small.

The complexity of the microcode control unit and the use of quantitative analysis drove RISC advocates to favor a push to software. Software prices did not become exorbitant, as was believed, and so the main reasons for maintaining the complexity of hardware were due to RAM prices. RISC advocates believed that a decrease in the clock cycles per instruction would more than pay for the increased dependence on RAM.

The main features of RISC architecture are the following:
- Emphasis on software.
- Only load and store may reference memory.
- Few instructions and addressing modes.
- Fixed instruction formats.
- Multiple register sets.
- Single clock cycle instructions.
- Hardware executes instructions.
- Complexity is in the compiler.
- Highly pipelined.
- Program code size large.
6.0 Similarities between CISC and RISC Architectures

In spite of their differences, the RISC and CISC architectures have come to employ many of the same techniques over the years. These features are commonly used because they cater to the overall goal of processor design that is increasing performance. The price both architectures pay is complexity. RISC systems have left some of their traditional design principles, and CISC systems have embodied some features originally reserved only for RISC designs. In this way their architectures are evolving towards each other.

Both systems now almost universally employ on-chip primary caches for faster memory access. This small amount of added complexity adds a lot in terms of overall data throughput. They have also added more registers for temporary storage due to increased use of the register-to-register addressing mode. The following items also can be seen on most CISC and RISC systems today, and the complexity they add to the core processor can be seen as a shifting of the burden onto hardware:

1. More functional units for pipelining
2. Additional instructions used by the new units
3. An increased pipeline depth
4. Branch prediction

7.0 Conclusion

From the architectural changes seen throughout the past two decades, it is obvious that the reasons spurring CISC researchers to reduce software size and complicate hardware are now out of context. Memory is inexpensive and continually dropping in price. RAM and secondary storage are practically taken for granted; Microsoft, for one among many other software companies, can attest to the fact that program size is not a real concern. In addition compilers have recently made great leaps. Direct manipulation of code in assembly language is
less often used now that compilers can regularly optimize High Level Language code in an automated way.

CISC designers have gradually embodied more RISC like design principles. At the general level, although not throwing out functional units, designers are looking to include components that will fully utilize what is already present. Intel and Hewlett Packard have migrated to a common instruction set architecture, creating smaller instructions and a more concise instruction set. Intel’s new architecture demonstrates that compilers do matter, because it fully depends on one to handle instruction execution order; it has abandoned the use of out-of-order execution in hardware. This change shows that CISC systems are shifting some complexity to software. Intel also features a shorter pipeline and lower clock cycle. CISC data paths have supported other RISC-like features, such as register-to-register addressing and an expanded register count. AMD’s Athlon chip even uses a combination of micro code and direct execution to handle control. CISC designs have not been the only things to change, also the classic RISC architecture has seen similar trend shifts, but towards more hardware complexity. Recent designs have increased the depth of the pipeline and implemented super scalar pipelining and dynamic branch prediction. These two components add a tremendous amount of hardware and complexity that were never found in original systems. In addition the instruction set has found more instructions to handle more complicated routines.

It is becoming difficult to distinguish a CISC processor from a RISC one; both have incorporated many similar features and functional units aiming to capture more performance from their CPUs. Even the two instruction sets are becoming more similar. The two architectures are evolving towards one another by incorporating these features, and their primary differences are due to their history. Both CISC and RISC designers share the common goal of increasing performance at a time when memory is widely available and compilers complement the processor. Both camps of designers have been adding functionality to decrease the CPI, once primarily a RISC concern. For now these
architectures can be differentiated by the terms CISC and RISC, but common future goals are sure to bring their designs closer together.
References

3. www.sunderland.ac.uk/~ts0jti/comparch/ciscrisc.htm
4. www.eg.bucknell.edu/~bsprunt/classes/elec_443_fall_2002/Lectures/elec_443_02_RISC_vs_CISC.pdf