

# **RISC++ Instruction Set Architecture**

A Hybrid ISA for Scalar and Vector Operations

J. Thomas Dunkerton

# RISC++ Instruction Set Architecture

## A Hybrid ISA for Scalar and Vector Operations

J. Thomas Dunkerton

This book is for sale at <http://leanpub.com/riscpp>

This version was published on 2020-04-22



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 J. Thomas Dunkerton

*Dedicated to my dear wife Ann, who has inspired me to publish this book, even though she has no clue what it is about. Thanks for believing in me and encouraging me to get it done already!*

# Contents

<b>About the Author</b> . . . . .	<b>1</b>
<b>Preface</b> . . . . .	<b>2</b>
<b>What is the RISC++ Architecture?</b> . . . . .	<b>4</b>
Hybrid Architecture for Scalar and Vector Operations. . . . .	4
Coprocessor Optimized for Scientific Programming . . . . .	5
RISC++ Core . . . . .	6
Dynamically Typed Scalar and Vector Operands . . . . .	10
RISC++ Conditional Instructions . . . . .	13
Vector Control Unit (VCU) . . . . .	17
What's so RISCy about RISC++? . . . . .	18
<b>High Performance Serial Processing</b> . . . . .	<b>20</b>
Instruction Level Parallelism (ILP) . . . . .	21
Out of Order Execution (OoOE) . . . . .	24
<b>Why Do We Need Another ISA?</b> . . . . .	<b>27</b>
RISC++ instruction set is simpler than X86 . . . . .	27
X86 Architecture Overview . . . . .	28
X86 Single Instruction Multiple Data (SIMD) . . . . .	29
A brief History of RISC++ . . . . .	32
Intel's oneAPI and Data Parallel C++ . . . . .	34

# About the Author

I was born in Nairobi, Kenya the son of missionaries. I spent most of my childhood in Kenya and Tanzania until I graduated from Rift Valley Academy and returned to the US for college. I graduated from Geneva College in 1978 with a double major in Physics and Computer science. I was very fortunate to get a job with IBM in Endicott N.Y. where I worked as a Computer Engineer testing the design of mid-range mainframes based on the 370 Architecture. A large part of my job involved writing software which verified that the hardware worked according to specifications.


In 1986 I resigned from IBM and went to Seminary to become a pastor. I also supported myself in doing free lance software development, as a substitute teacher and working in the mental health field. I found a way to keep myself up to date on new technologies by studying new developments in computer architectures. I decided to design an Instruction Set Architecture (ISA) based on the C++ language. To do this, I have spent countless hours on the internet learning about various architectures that have since come and gone. I have spent much time filling my notebooks with various ideas.

This has become an obsession that occupies a part of my brain that would have otherwise become atrophied. But the computer industry kept evolving and my designs kept becoming obsolete. Add to that my desire to be perfect and the work never got done. Finally, now that I am retired, I have time to get it done. It is far from perfect, but it is as ready as it is going to be. I would value any input on how to make it better.

In my retirement I continue to work as a part-time substitute teacher, who loves gardening, hiking, but most of all playing with my grandchildren. I am a Star Trek fan and I like to binge-watch all the series on Netflix. My favorite is "Enterprise" (I know, I'm weird) and my second favorite is "Deep Space Nine". I enjoy board games, but I'm not as much of a fanatic as my wife. For music, I enjoy the "oldies but goldies", like Simon and Garfunkle, Kenny Rogers, Johnny Cash, John Denver, Peter Paul, and Mary, but also Scott Joplin and Classical. But my favorite music of all is the old Hymns. Basically I'm a nerd who is still stuck in the sixties.

I really don't want anyone to steal or take credit for my work. Neither do I want to take credit for the work of others. I value the input of others, and I want these ideas discussed and evaluated. I would especially appreciate input from hardware designers, compiler writers, and people who write performance critical code. Please be kind. This is my baby.

# Preface

This book is a proposal for an [Instruction Set Architecture \(ISA\)](#)<sup>1</sup> which is designed to run a single execution thread for both scalar and vector operations. The RISC++ Architecture (Reduced Instruction Set for C++) is designed to have a close semantic fit with high-level programming languages such as C++<sup>2</sup>. 

One of the goals of RISC++ is to simplify both hardware and software design. By designing the machine language instruction set with high-level languages in mind, the binary code generated will be more efficient in both memory space and execution time. The RISC++ instruction set often has a one to one correspondence to C++ operators, data types, and expressions.

It has taken me many years to come to the point of publishing this book. It began as a tiny project which has grown over the years because the industry has evolved and I was always trying to catch up. I had no deadlines, I am an obsessive perfectionist, and I do tend to procrastinate. So now that I am retired I finally have time get it done.

I used to have great dreams of getting a patent, selling it to Intel®, IBM® or whoever might want to buy it, and live my life in luxury. But then I would have to keep it a secret and I couldn't get input from other people. So now I've decided that this copyrighted book will be enough proof that these are my ideas and I can put the question of patents off for another day. If I can sell some copies and use this to educate others, then that is enough, for now.

The target audience of this book is people who are familiar with C++, assembler level programming, and binary machine code. It is also important to have some understanding of micro-processor design. As we go through the book I will try to explain new concepts as they come up. Before getting into the details of RISC++ I will give a very brief summary of the history of micro-processors and the instruction sets that run on them.

The bulk of information will be about the very large family of processors developed by many different manufacturers known as X86<sup>3</sup>. Intel and AMD (Advanced Micro Devices) have been primarily responsible for designing X86 Instruction Set Architecture and its many extensions. We will review the history of this ISA over time to illustrate how RISC++ is simpler.

For those who are unfamiliar with microprocessors, and those who want a quick refresher, I highly recommend Jonathan Stokes' book which can be purchased on Amazon: [Inside the Machine, An Illustrated Introduction to Microprocessors and Computer Architecture](#)<sup>4</sup>.

A good online overview is [Modern Microprocessors - A 90-Minute Guide](#)<sup>5</sup> by Jason Robert Carey Patterson. This covers a wide range of topics in a short amount of time. It is highly recommended.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Instruction\\_set\\_architecture](https://en.wikipedia.org/wiki/Instruction_set_architecture)

<sup>2</sup><https://en.wikipedia.org/wiki/C%2B%2B>

<sup>3</sup><https://en.wikipedia.org/wiki/X86>

<sup>4</sup><https://www.amazon.com/Inside-Machine-Introduction-Microprocessors-Architecture/dp/1593276680>

<sup>5</sup><http://www.lighterra.com/papers/modernmicroprocessors/>

At the time of the publication of this book, there is no hardware which implements the RISC++ Architecture. I claim the ideas in this book as my personal intellectual property. I ask that the copyright be honored and that I be given credit for any of the ideas that are original to me. I will also make every effort to credit the works of others. I want this document to be used for educational purposes and I want the ideas to be as wide spread as possible.

I hope to stimulate discussion and get ideas for improving the design. Many of the ideas have come through my own study using internet resources. I will do my best to provide links where people can study these ideas on there own. I use Wikipedia extensively, but I caution that these articles cannot necessarily be considered authoritative, but only as a starting point for further study. After reading this book and following the links, may I suggest that you make a donation to Wikipedia for providing such a rich set of relevant online information.

I would welcome comments and suggestions from experts in various fields. I would greatly appreciate input from people who design compilers for various languages such as C++. I would also welcome comments from hardware designers, from operating system experts, and from people who design critical performance scientific software. I am not an expert in any of those fields, but I have spent a great deal of time studying various computer architectures both at the ISA level and the implementations in hardware. I have built on top of what others have done and tried to streamline and pick the best features.

After reading this book I would encourage people who feel so inclined to:



1. Offer suggestions for improvement of the instruction set or the hardware.
2. Help develop an “official” standardized definition of the RISC++ ISA.
3. Work cooperatively to develop the following tools:
  - A macro Assembler to generate RISC++ binary code.
  - A C++ compiler which generates RISC++ Assembler code.
  - A RISC++ emulator, with debug capability.
  - A new C++ math library optimized for RISC++.
4. Develop Verilog code to implement the RISC++ hardware.
5. Run Verilog code on a Field Programmable Array.
6. Try to convince Intel®, IBM® or AMD®, to build a RISC++ CPU.

Even if the RISC++ Architecture is never implemented in hardware, the tools might be a way of educating people on compiler and hardware design. I would hope the software would be open source and used in the education community. In fact, I would even dare to presume that this book could be used in college courses about compiler design or other related topics.

# What is the RISC++ Architecture?

The RISC++ Architecture began as a hobby in which I was trying to define a binary machine code and the associated Assembler language which would have a close “semantic fit” with the C++ language. From the very beginning, RISC++ had the concept that the data type of an operation was to be defined by the *registers* rather than the *instructions*. Then the RISC++ instructions would apply to a variety of data types and combinations of types. This has never changed. Unlike most Instruction Set Architectures (ISAs), a single ADD instruction applies to various data types like *float*, *double* and *int*. This is facilitated by **Type/Status Registers (TSRs)** which will be explained shortly.

Another constant in RISC++ has been the use of *conditional instructions* which are either executed or skipped depending if previously set *condition bits* are either true or false. Not only are the BRANCH instructions conditional, but so are instructions like ADD, SUBTRACT, MULTIPLY and DIVIDE. This means that many branches can be eliminated from the instruction stream thus streamlining the binary code. Branches and branch prediction have always been a challenge for hardware designers. It is my belief that the RISC++ methodology will both simplify and accelerate the machine code.

A recent innovation in RISC++ is the introduction of *variable length typed Vectors*. The RISC++ vector instructions, like the the Scalar counterparts, use generic instructions such as VECTOR ADD, VECTOR MULTIPLY. The data type is associated with the Vector data in a **Type/Status/Length Registers (TSLRs)**.

Tightly coupled to the Vector instructions and typed data is the concept of the **Typed Pointer (TPTR)**. Typed Pointers carry the data type and length in the upper 16 bits of a 64-bit register while the lower 48 bits carry the memory address of the data. Typed pointers are similar to *reference variables* in C++ and are very important in Vector operations.

This chapter will introduce the main features of the RISC++ architecture, while later chapters will supply greater detail.

## Hybrid Architecture for Scalar and Vector Operations.

RISC++ is a hybrid architecture designed to enable fast execution of serial instruction streams which include both scalar and vector data. It incorporates the features of both [Central Processing Units \(CPU\)](https://en.wikipedia.org/wiki/Central_processing_unit)<sup>6</sup> and [General-purpose computing on graphics processing units \(GPGPU\)](https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units)<sup>7</sup>. RISC++ is not meant to replace various CPUs, such as the Intel X86<sup>8</sup> family of processors, but to coexist with them.

---

<sup>6</sup>[https://en.wikipedia.org/wiki/Central\\_processing\\_unit](https://en.wikipedia.org/wiki/Central_processing_unit)

<sup>7</sup>[https://en.wikipedia.org/wiki/General-purpose\\_computing\\_on\\_graphics\\_processing\\_units](https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units)

<sup>8</sup><https://en.wikipedia.org/wiki/X86>



When I speak of the X86 I am referring to a whole family of processors that began with the original Intel® 8086™ and have grown and expanded over the years. The X86 Architecture has evolved from 16 bits to 64 bits. The X86 ISA has been implemented on many chips designed by various companies such as Intel®, AMD®, IBM® and many others. There is no need to compete with or replace either the X86 ISA or the chips that implement it. Instead the RISC++ is intended to be a coprocessor that can offload computation heavy workloads.

Because of the support for 256 byte long vectors, a RISC++ core would be larger than today's X86 cores. But a single core would be considerably smaller than a GPU. Since the RISC++ cores are relatively small by today's standards, several (2, 4, 8) of them can be put on a single chip. This would make a really nice laptop for a scientist who needs fast computing in the field.

Today's existing math libraries already manage many parallel threads of operation each working on a part of the problem. These are solutions that have been well developed and are already supported by various C++<sup>9</sup> code libraries. Since RISC++ binary code will have a one to one correspondence to many C++ operations, those libraries can be recompiled into code with a smaller foot print, and faster path lengths.

Once a new compiler is written to target the RISC++ architecture, these libraries should be able to be ported quite easily. Since much of Linux is written in C++, it should also port easily. The Assembler portions of a RISC++ based Linux should also be intuitive with good control over the machine code.

## Coprocessor Optimized for Scientific Programming

The decision to introduce a new architecture should never be taken lightly. Many well designed and well-financed processors have failed in the marketplace. Meanwhile, legacy architectures like the X86 continue to dominate. The reason is simple. Software development is costly. Hardware that can run legacy software without recompilation will always be very popular especially in the PC market.

Furthermore, ramping up production of new chips is extremely expensive. The PC market has no need to move away from the dominance of Microsoft Windows and/or the X86 Architecture. Even people who prefer Linux based PCs and Laptops have great X86 based hardware to run it on with all the integrated graphics. RISC++ isn't for phones or anything that needs an integrated graphics processor. RISC++ is designed to coexist as a scientific coprocessor, rather than to replace X86.

The RISC++ chip is designed to enhance scientific programming in a variety of environments:

1. As a coprocessor on the same motherboard as an X86 chip.
2. As a standalone computer running an Operating System like Linux.
3. As a processor mounted in a tower, on a PCI card.
4. As a computer "blade" mounted in a supercomputer system.
5. As a processor in a high-performance gaming machine with a Graphics Coprocessor.
6. As a high performance laptop computer designed for scientists working in the field.

---

<sup>9</sup><https://en.wikipedia.org/wiki/C%2B%2B>

RISC++ may be used in standalone applications that do not require X86 compatibility, such as Linux, or other Operating System environments. These would probably be running in Console mode since that is still very popular in many Linux server environments. If high-resolution graphics are required the system may need a dedicated graphics processor which are readily available on PCI cards. A Linux based laptop could also be built with a RISC++ chip and a separate GPU chip. The space that is often devoted to high-speed graphics on modern X86 chips is devoted to scientific vector processing on a RISC++ chip.

The niche for this architecture is the scientific computing community, though some business applications might benefit as well. RISC++ cannot replace GPGPU applications when it comes to extremely high performance parallel processing of very large data sets. Instead the architecture integrates vector instructions into the serial instruction generated by the compiler.

X86 processors also do this in a limited way using [Single Instruction, Multiple Data \(SIMD\)](#)<sup>10</sup>. The X86 SIMD instruction set has evolved over the years leaving us with a complex array of instructions for compiler writers to choose between. RISC++ starts over with a new set of *Vector instructions* which work well with the standard “for loop” which is found in so many computer languages.

## RISC++ Core

The RISC++ hardware consists of several “cores” which can be mounted on a chip. Each core supplies the components necessary to run a “thread” of operation. A thread is a serial instruction stream consisting of both scalar and vector instructions. This serial stream of instructions is converted at run time into several parallel instructions. Each core interfaces to three memory caches:

### Instruction Cache

- Direct mapped read only cache with fixed length buffers.
- Connected to Instruction Control Unit (ICU)
- Feeds instructions to be decoded and dispatched.

### Scalar Data Cache

- Set associated read/write cache with variable length data access.
- Connected to Fast Integer and Typed Scalar Units.
- Variable data length from 8 to 128 bits.

### Vector Data Cache

- Direct mapped read/write cache with 256 byte buffers.

---

<sup>10</sup><https://en.wikipedia.org/wiki/SIMD>

- Connected to Vector Execution Unit
- Variable data length from 1 to 256 bytes.

The instructions are dispatched to “Execution Units” where they perform specialized operations:

### **Instruction Control Unit (ICU):**

- Fetches Instructions from fixed length Instruction Cache.
- Decodes and dispatches instructions to other Execution Units.
- Handles Branches, Interrupts and Conditional Instructions.
- Manages Out of Order Instruction Execution (OoOE).
- Instruction pointer defines next instruction to be executed.
- Contains the following registers:
  - a. 16 x 64-bit Address Registers
  - b. 32 x 16-bit Short Integer Registers.
  - c. 64 x 8-bit Type Status Registers (TSR).
  - d. 16 x 1-bit In Order Condition Bits.
  - e. 16 x 1-bit Out of Order Condition Bits.

### **Load/Store Unit (LSU):**

- Moves data back and forth between data caches and register banks.
- Reads and writes 8, 16, 32, 64 or 128 bits to/from FIU and TSU
- Reads and writes 1 to 256 bytes to/from Typed Vector Unit (TVU).
- Saves and Restores various Register banks.
- Converts Data Types to/from memory format and register format.
- Manages PUSH and POP to various Stacks.

### **Fast Integer Unit (FIU):**

- Used for pointers, control registers and index registers.
- Executes Integer data in a single cycle.
- 32 x 64-bit integer registers.
- Also accesses various registers in ICU.
- No multiply and divide or other multicycle instructions.

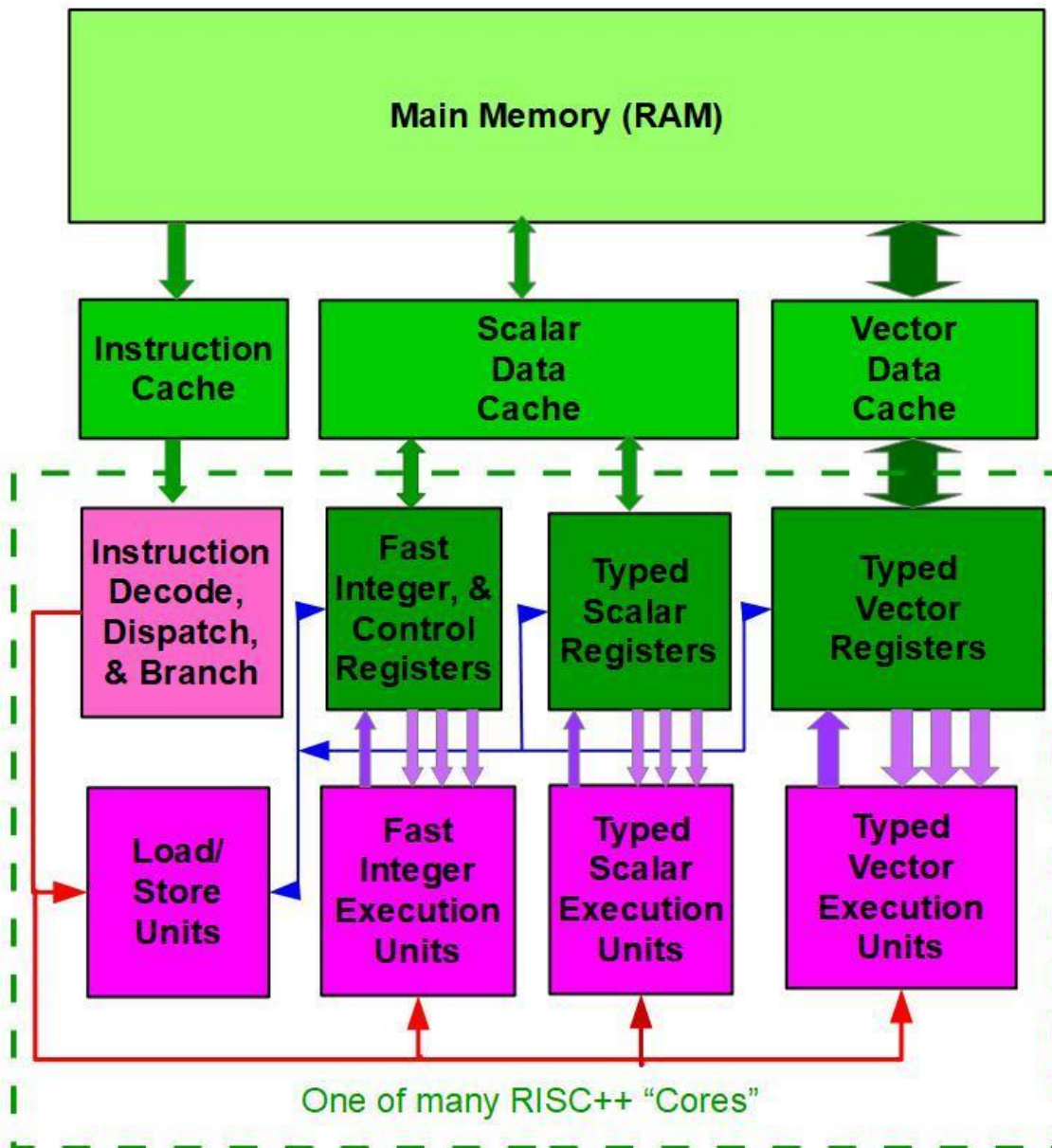
## **Typed Scalar Unit (TSU):**

- Used for both Integer and Floating Point Data.
- TSRs allow generic instructions which use all data types.
- TSU uses Out of Order Execution.
- 64 x 64-bit Data registers
- 128-bit data in two registers.
- Each register is associated with an 8-bit (TSR) in ICU.
  - a. 4-bit Data Type Codes.
  - b. 4-bit Status Code.

## **Typed Vector Unit (TVU)**

- 16 x 16-bit Type/Status/Length Registers (TSLR)
  - a. 4-bit Data Type Code.
  - b. 4-bit Status Code.
  - c. 8-bit Vector Length.
- 16 x 256-Byte Variable length Vectors:
- Each Vector is mapped to 16 x 128-bit parallel registers.
- Same data types as Typed Scalar Unit.
- Supports C++ standard string operations (8 or 16-bit).

## RISC++ Hardware Overview



## Dynamically Typed Scalar and Vector Operands

The reason RISC++ was invented was to make assembler code and the resulting machine code, very close semantically to the C++ language. This was done to simplify both the machine code generated and the underlying hardware. Dynamically typed registers are the very heart of RISC++ and the principle way in which C++ semantics are built into the Assembler and machine code.

High level languages (including C++) first declare the data as a given type then have generic operators (like + - \* / %) that operate on all different data types. That is exactly what RISC++ does. For example, a single ADD instruction may use integer or floating point operands of various lengths. The compiler does not need to generate different machine language instructions for each data type. There just only one instruction ... ADD. The same is true for all the standard C++ operators. Furthermore, source operands of different types can be freely intermixed, in many cases, just as they are in C++.

Vector operations use 16 bit *Type/Status/Length Registers (TSLR)* which define variable length typed vectors from 1 to 256 bytes in length. The vector “registers” (or buffers) are 256 bytes long as opposed to the Intel SSE family which uses 128 bit registers.

Because they are variable length, they can be used for string operations. Like the Scalar Registers, they carry the data type, status and length in a 16 bit TSLR. These 256 byte registers are nowhere near the size of parallel data that can be processed in today’s Graphics Processing Units. But for smaller vectors they are more powerful than X86 cores.

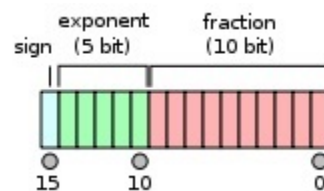
Register	4 bit Field	4 bit Field	8 Bit Field
TSR	Type code	Status code	
TSLR	Type code	Status code	Vector Length

## RISC++ Data Types

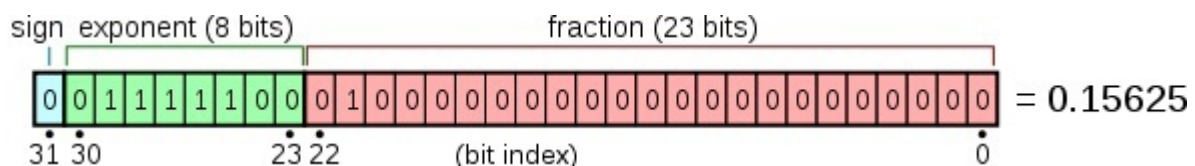
RISC++ defines the following 4-bit data type codes in Hexadecimal:

Code	Symbol	Description	Max Vector Size
x0	SWI	Software Interrupt	256 Bytes
x1	BCD	32 x 4 bit Binary Coded Decimal	16 x 128 bit
x2	BYTE	16 x 8 bit String of Bytes	256 x 8 bit
x3	DF128	128 bit Floating Point Decimal	16 x 128 bit
x4	BF16	16 bit Binary Floating Point	128 x 16 bit
x5	BF32	32 bit Binary Floating Point	64 x 32 bit
x6	BF64	64 bit Binary Floating Point	32 x 64 bit
x7	BF128	128 bit Binary Floating Point	16 x 128 bit
x8	IS16	16 bit Integer Signed	128 x 16 bit
x9	IS32	32 bit Integer Signed	64 x 32 bit
xA	IS64	64 bit Integer Signed	32 x 64 bit
xB	IS128	128 bit Integer Signed	16 x 128 bit
xC	IS16	16 bit Integer Unsigned	128 x 16 bit
xD	IS32	32 bit Integer Unsigned	64 x 32 bit
xE	IS64	64 bit Integer Unsigned	32 x 64 bit
xF	IS128	128 bit Integer Unsigned	16 x 128

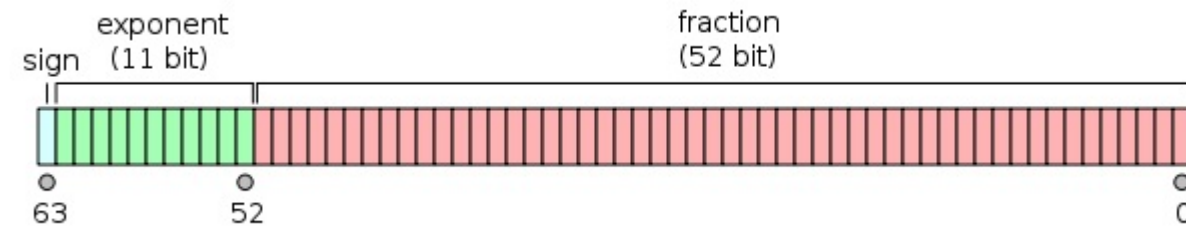
## Floating-Point Arithmetic (IEEE 754) Binary Floating Point data formats:



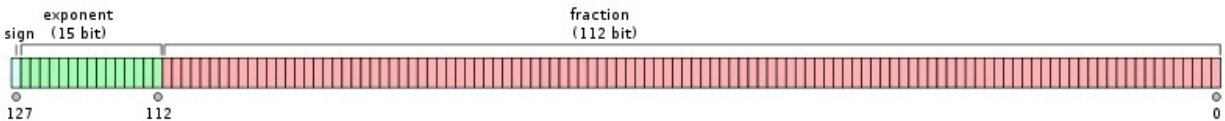
16-bit Binary Floating Point



32-bit Binary Floating Point



64-bit Binary Floating Point



128-bit Binary Floating Point

## Intermixing of Input Operands of Different Types

All of the typed registers are 64 bits long. Data is converted from the memory format to the register format during the LOAD. Then it is converted back to the memory format during the STORE. This is true for both integer and floating point data.

When loading signed integer data, the sign bit is propagated to the upper bits in the target register. The TSR is then flagged with the data type from memory. Similarly, when loading unsigned integer data, the upper bits in the target register are filled with zeros.

The two integer source operands may have different signs and lengths of data. The target register inherits the length of the larger input operand. For example, if the two source operands are IS16 and IS32, the target operand would be IS32. If both source operands are unsigned, the target will be unsigned. If one of the input operands is unsigned, and this other is signed, the target operand will be signed.

Typed data instructions may intermix integer data types of different lengths and signs without errors. The data will overflow and wrap according to the data type in the TSR. However, if an attempt is made to mix two input operands in which one is *integer* and the other is *floating point*, the status field of the target register will indicate a “Data Type Error”.

When we get to the section on interrupts I will expand on this further. To avoid flagging an error, the compiler or the Assembler programmer must insert a CONVERT instruction which will convert the integer operand into a floating point, and then issue the instruction.

When typed registers are to be used for floating point operations, the internal registers are in the BF64 format (C++ type double). When the memory location is BF16 (half float) or BF32 (float), the data is converted to the BF64 format during the load. The data type of the memory location is then flagged in the TSR for rounding and overflow purposes. Input operands of BF16, BF32 and BF64 can be freely intermixed. When a target register is written, it *inherits* the type of the input operand of higher precision.



128 bit scalars use two even/odd registers. **All loads of 128 bit data must specify an even register.** Input data operands of 128 bit length *must match* data types. The type field of the odd register is set to the 128 bit type and the status is set to 1111 (odd register). If an Assembler programmer forgets and references the odd register it will produce a data type error. Presumably the compiler should know better.

## Example 1 - Intermixing *float* and *double* operands:

Consider the following C++ code:

```
1      1. float  a = 5.5;
2      2. double b = 3.5;
3      3. float  c = a + b;
```

The resulting RISC++ Assembler code would be:

```
1      1. DM      a,      BF32 = 5.5;    // Define Memory a
2      2. DM      b,      BF64 = 3.5;    // Define Memory b
3      3. DM      c,      BF32 = 0;      // Define Memory c
4      4. LD      R4,      BF32,  a;      // Load a into R4, flag as float
5      5. LD      R5,      BF64,  b;      // Load b into R5, flag as double
6      6. ADD     R6,      R4,      R5;    // R6 = R4 + R5, R6 is now double
7      7. ST      R5,      BF32,  c;      // Store R5 in c, Convert to float
```

## RISC++ Conditional Instructions

Another way in which RISC++ code mimics the C++ language is with *conditional instructions*. These instructions allow “IF THEN ELSE” clauses to be encoded without BRANCH instructions. The *Condition Bits (CBs)* are set various COMPARE instructions.

The Instruction Control Unit (ICU) examines the status of the condition bit then dispatches the instruction to an *Execution Unit (EU)* if the bit is true. Otherwise the instruction is skipped and eliminated from the instruction stream. Condition bit number 0 is always 1 and if the condition bit field is 0, the instruction will be interpreted as unconditional.

There are six compare instructions which encode (==, !=, <, <=, >, >= ) instructions:

```

1      1. CMPEQ    CB,  RX,  RY;    // Set CB if RX == RY
2      2. CMPNE    CB,  RX,  RY;    // Set CB if RX != RY
3      3. CMPLT    CB,  RX,  RY;    // Set CB if RX <  RY
4      4. CMPLT    CB,  RX,  RY;    // Set CB if RX <= RY
5      5. CMPGT    CB,  RX,  RY;    // Set CB if RX >  RY
6      6. CMPGE    CB,  RX,  RY;    // Set CB if RX >= RY

```

If the condition bit is TRUE, the instruction is dispatched to an execution unit. If the condition bit is FALSE, the instruction is skipped in the ICU and eliminated from the instruction stream. The Condition Bits can be CB manipulated by CB logical instructions:

```

1      1 CBAND     CB,  CBX,  CBY;    // CB = CBX && CBY, Logical AND
2      2 CBOR      CB,  CBX,  CBY;    // CB = CBX || CBY, Logical OR
3      3 CBNOT     CB,  CBX;          // CB = ! CBX,      Logical NOT

```

## Example 2 - Conditional Move

Consider the following C++ code:

```

1      1. int      a = 10;
2      2. int      b = 15;
3      3. int      c = 0;
4      5. if a < b then c = a;
5      6. if a > b then c = b;

```

The resulting RISC++ assembler code would be:

```

1      1. DM       a,      IS16 = 10;    // Define memory a
2      2. DM       b,      IS16 = 15;    // Define memory b
3      3. DM       c,      IS16 = 0;     // Define memory c
4      4. LD       R4,     IS16,  a;      // Load a into R4, flag as int
5      5. LD       R5,     IS16,  b;      // Load b into R5, flag as int
6      6. CMPLT    CB4,    R4,      R5;    // set CB4 if R4 < R5
7      7. CMPGT    CB5,    R4,      R5;    // set CB5 if R4 > R5
8      8. CMOV     R6,     R4,      CB4;   // if CB4 is true R6 = R4
9      9. CMOV     R6,     R5,      CB5;   // if CB5 is true R6 = R5
10     10. ST      R6,     IS16,  c;      // store R6 into c

```

## Example 3 - Conditional Arithmetic

The following example puts together the *dynamic typing of register data*, the use of COMPARE instructions, and the use of *conditional arithmetic instructions*, to encode an “IF THEN ELSE” clause without any BRANCH instructions:

Consider the following C++ code:

```

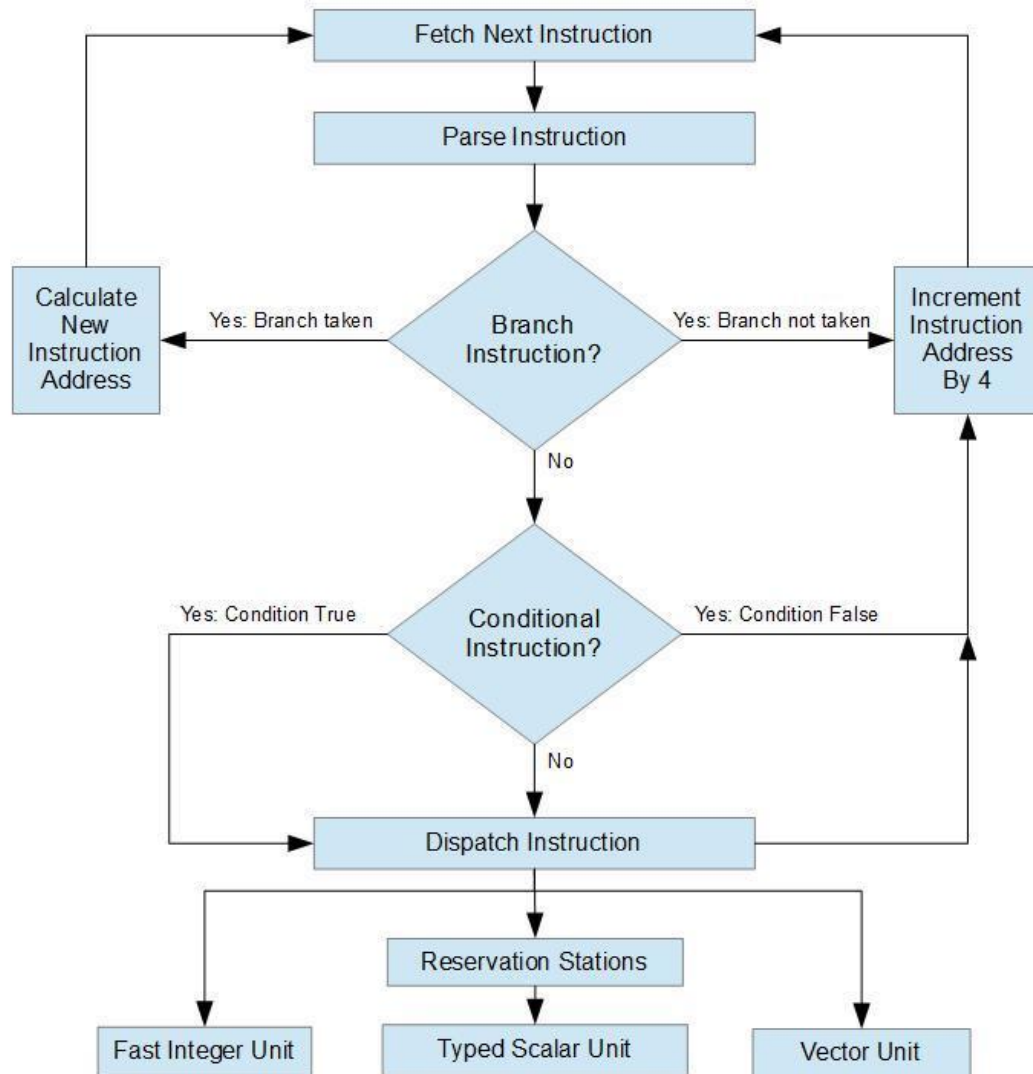
1      1. float    a = 4.0;
2      2. double   b = 16.0;
3      3. float    c = 12.0;
4      4. float    d = 0.0;
5      5. double   e = 0.0;
6      6. if (a + c) == b then {
7          7.      d = a * a;
8          8.      e = c - a;
9      9.      else
10     10.      d = b / a;
11     11.      e = c + a; }
```

The resulting RISC++ assembler code would be:

```

1      1.  DM      a,      BF32 = 4.0;    // Define Memory a
2      2.  DM      b,      BF64 = 16.0;   // Define Memory b
3      3.  DM      c,      BF32 = 12.0;   // Define Memory c
4      4.  DM      d,      BF32 = 0.0;    // Define Memory d
5      5.  DM      e,      BF64 = 0.0;    // Define Memory e
6      4.  LD      R4,      BF32,  a;      // Load R4, flag as float
7      5.  LD      R5,      BF64,  b;      // Load R5, flag as double
8      6.  LD      R6,      BF32,  c;      // Load R6, flag as float
9      7.  LD      R7,      BF32,  d;      // Load R7, flag as float
10     8.  LD      R8,      BF64,  e;      // Load R8, flag as double
11     9.  ADD      R9,      R4,      R6;   // R9 = R4 + R5, R9 inherits type double
12    10.  CMPEQ    CB4,      R9,      R5;   // if R9 == R5 set CB4 to "true"
13    11.  CBNOT    CB5,      CB4;         // Set CB5 as NOT of CB4 (else clause)
14    12.  MUL      R7,      R4,      R4;    CB4; // if CB4 is true, R7 = R4 * R4
15    13.  SUB      R8,      R6,      R4;    CB4; // if CB4 is true, R7 = R6 - R4
16    14.  DIV      R7,      R5,      R4;    CB5; // if CB5 is true, R7 = R5 / R4
17    15.  ADD      R8,      R6,      R4;    CB5; // if CB5 is true, R8 = R6 + R4
18    16.  ST       R7,      BF32,  d;      // Store R7 (16.0), convert to float
19    17.  ST       R8,      BF64,  e;      // Store R8 (8.0) as double
```

## RISC++ Instruction Flow



RISC++ Instruction Flow

## Vector Control Unit (VCU)

I'm not going to spend much time on describing the VCU in this chapter. I will devote a whole chapter to that. For now I just want to make a few points and give an example. The VCU depends very much on the concept of the “*Typed Pointer*”. The typed pointer is a 64-bit Fast Integer Register with the following format:

Register	4 bit Field	4 bit Field	8 Bit Field	48-bit Field
TPTR	Type code	Status code	Vector Length	Memory Address

The TPTR is a “*reference variable*” that can be passed on a stack. When the length field is x00 the data type is scalar, and can be used to load scalar registers and set the TSR. The TPTR is used to load a 256 byte buffer from the vector cache into the 16 x 128 bit parallel registers that make up a 256 byte long *Vector*. The upper 16 bits of the TPTR can also be supplied by a 16-bit Short Integer Register. This is useful for dynamically updating the data length in “for loops”.

### Example 4 - Vector Addition

The following example shows how to ADD two Vectors and put the results in a third. The example assumes that all three vectors are exactly the maximum length of 256 bytes. Real world problems would not have exactly 256 byte long vectors. But this example is useful for illustrating the main points:

Consider the following C++ code:

```

1      1. float  VA [64];          // 256 bytes is 64 floats
2      2. float  VB [64];
3      3. float  VC [64];
4      5. for   (i = 0; i < 64 ; i++)
5      6.      VC [i] = VA [i] + VB [i];

```

The resulting RISC++ Assembler code would be:

```

1      1.  DM    VA,      BF32 [64];      // Define Vector Memory VA
2      2.  DM    VB,      BF32 [64];      // Define Vector Memory VB
3      3.  DM    VC,      BF32 [64];      // Define Vector Memory VC
4      4.  ORI   ISR4,    0,      x50FF;   // ISR4 used as TSLR
5      5.                                     // Type 5 (BF32), Length 256
6      6.  LPA   R7,      VA              // R7 is 48-bit Pointer Address VA
7      7.  LPA   R8,      VB              // R8 is 48-bit Pointer Address VB
8      8.  LPA   R9,      VC              // R9 is 48-bit Pointer Address VC
9      9.  LDV   V2,      R7,      ISR4;   // Load Vector 2, 256 bytes from VA
10     10. LDV   V3,      R8,      ISR4;   // Load Vector 3, 256 bytes from VB

```

```

11      11.  ADDV   V4,      V2,      V3;      // VR4 = VR2 + VR3, Add parallel vectors.
12      12.  STV    V4,      R9,      ISR4;     // Store 256 Bytes into VC location

```

Looking at the Assembler code may give the impression that there is a lot of overhead. In reality, the setting up of vector addresses and types will usually take place outside a “for loop”. The actual code inside the loop may be quite efficient. Vector manipulation can be complex but there are many ways to deal with these complexities as we shall see in a later chapter.

## What’s so RISCy about RISC++?

To end this chapter, I want to discuss if RISC++ is really a Reduced Instruction Set Computer in the classic sense.

There are two common types of Instruction Set Architectures (ISAs), the [Reduced Instruction Set Computer \(RISC\)](https://en.wikipedia.org/wiki/Reduced_instruction_set_computer)<sup>11</sup> and the [Complex Instruction Set Computer \(CISC\)](https://en.wikipedia.org/wiki/Complex_instruction_set_computer)<sup>12</sup>. In the early days of computing, when memory was very expensive, the early ISAs were designed to squeeze as much functionality as possible into every instruction. Also, the instructions were designed to have a close “*semantic fit*” with High Level Languages.

But the instruction sets were difficult to implement in hardware. The RISC concept was introduced to make hardware design simpler at the cost of a larger memory foot print. Complex operations were moved to software rather than hardware. Then the hardware designers could focus on doing what hardware does best, and leaving the more difficult tasks to software.

The original definition of a RISC processor was:

1. Fixed length instructions to make fetch and decode more efficient.
2. All arithmetic and logical (ALU) instructions are register to register.
3. Instructions complete in a single cycle.
4. Simple memory addressing modes.
5. More registers than CISC, (often 32 instead of 8 as in early X86).

At first RISC processors were indeed simple and fast. Many of them were (and still are) used as micro-controllers, I/O processors, etc.

But today, “under the covers” in both RISC and CISC processors is a very complex micro-architecture. We will be looking at that later. It has nothing to do with RISC vs CISC. Actually, at the “core”, all computers have a RISC architecture. Today, the X86 family of processors dynamically convert CISC instructions into an internal RISC like microcode at run time.

The Power PC family of processors have a RISC ISA which also runs on a very complex hardware micro-architecture. Both these micro-architectures have two things in common which account for their hardware complexity:

<sup>11</sup>[https://en.wikipedia.org/wiki/Reduced\\_instruction\\_set\\_computer](https://en.wikipedia.org/wiki/Reduced_instruction_set_computer)

<sup>12</sup>[https://en.wikipedia.org/wiki/Complex\\_instruction\\_set\\_computer](https://en.wikipedia.org/wiki/Complex_instruction_set_computer)

1. [Superscalar Processing](#)<sup>13</sup>
2. [Out of Order Execution \(OoOE\)](#)<sup>14</sup>

These work together to speed up the execution of sequential threads of instructions. I will devote a whole chapter to this subject. When it comes to the RISC vs CISC debate, the truth is CISC has “won”. This is not the result of technological matters but economics.

The ubiquity of Microsoft Windows running on X86 based hardware fed money into Intel’s coffers so that they could produce huge volumes of processors, which in turn drove down the cost. When Apple decided to port their operating system from the IBM Power PC to X86, it established Intel as the king of CPUs. The [Intel Core \(micro-architecture\)](#)<sup>15</sup> is the hardware that made that happen.

If you look at the five points and two sub-points about RISC above I will say that everything is true except for number “3”. RISC++ instructions do **not** complete in a single cycle. Neither, for that matter, do instructions in the Power PC RISC. Because of OoOE the Power PC had many instructions “in flight” at the same time and many of them took many cycles to complete. The same is true of RISC++.

The Typed Scalar Unit (TSU) operates on data with 15 different data types with some instructions that may take a very long time to complete. While some instructions such as ADD integer instructions may complete very quickly, others like DIVIDE 128-bit Binary Floating Point, take much longer. So will any instructions that depend on items of data that are not in cache. It may take many cycles to load the data from memory. This is why Out of Order Execution is so vital.

There are four innovations which are the *essence* of RISC++:

1. Generic Scalar instructions using Type/Status Registers (TSR)
2. Generic Vector instructions using Type/Status/Length Registers (TSLR)
3. Branch Elimination using Condition bits and Conditional Instructions.
4. Powerful Vector manipulation using Typed Pointers (TPTR)

We have already seen how these work together in the Assembler examples above. What we have not yet seen is how the status code is used to efficiently manage instructions in the reservation stations and how interrupts are handled. I just want to make the point now that because there is a status code on **all** the Typed Registers, it significantly reduces the complexity of how Out of Order Execution is managed.

---

<sup>13</sup>[https://en.wikipedia.org/wiki/Superscalar\\_processor](https://en.wikipedia.org/wiki/Superscalar_processor)

<sup>14</sup>[https://en.wikipedia.org/wiki/Out-of-order\\_execution](https://en.wikipedia.org/wiki/Out-of-order_execution)

<sup>15</sup>[https://en.wikipedia.org/wiki/Intel\\_Core\\_\(microarchitecture\)](https://en.wikipedia.org/wiki/Intel_Core_(microarchitecture))

# High Performance Serial Processing

One of the ways to speed up scientific computation is by using many processors, working in parallel, each working on a part of the problem. The kinds of problems that are best served by a *huge* degree of [Data parallelism](https://en.wikipedia.org/wiki/Data_parallelism)<sup>16</sup> are called “*embarrassingly parallel*” problems. Usually, this involves very large data arrays which are divided up among many processors all running the same program. However, most real-world problems have sections of code that **must** be computed in serial.

According to Amdahl’s law, “**the speedup of a program using multiple processors in parallel computing is limited by the time needed for the *sequential* fraction of the program.**”

A somewhat humorous illustration of this principle asks the question “if it takes nine months for one woman to have a baby, can nine women have a baby in one month?” The answer of course is no! Some tasks simply can’t be shared no matter how many willing volunteers you may have!

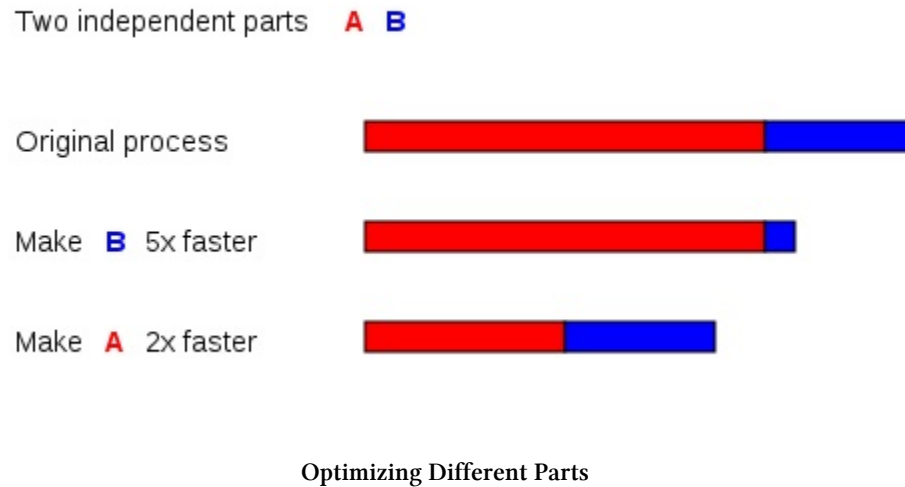
The same is true with real life problems we try to solve on a computer. There are some things that *must* be done in sequence because the results of the next operation depend on what has been done before. We call this “Serial” or “Sequential” programming. Each task must be completed before the next one can begin. But some very smart hardware designers have figured out ways to speed up sequential programs also, as we shall soon see.

While optimizing the parallel portion of a program through parallel processors is profitable, any acceleration of the sequential portions of code does even more to the speed up the overall program. In real life scientific applications, sequential code is needed to calculate data elements which will later be calculated in parallel using Vector operations. Any acceleration of the sequential part of the program yields significant acceleration of the overall program.

---

<sup>16</sup>[https://en.wikipedia.org/wiki/Data\\_parallelism](https://en.wikipedia.org/wiki/Data_parallelism)





Assume that a task has two independent parts, A and B. Part B takes roughly 25% of the time of the whole computation. By working very hard, one may be able to make this part 5 times faster, but this reduces the time of the whole computation only slightly. In contrast, one may need to perform less work to make part A perform twice as fast. This will make the computation much faster than by optimizing part B, even though part B's speedup is greater in terms of the ratio, (5 times versus 2 times)

Source: Wikipedia article on [Amdahl's Law](#)<sup>17</sup>:

## Instruction Level Parallelism (ILP)

To provide the best computation speeds possible for the sections of code that must be run in serial, we must find individual instructions that can be run in parallel. This is called [Instruction Level Parallelism \(ILP\)](#)<sup>18</sup>.

ILP facilitates the acceleration of a *single thread* of computation and takes place on a *single processor*. This is not the same as Data Level Parallelism (DLP) in which a large data set is divided up and the computation is spread over *many* processors.

The amount of ILP in any given application is determined by the algorithm. The key concept needed to understand ILP is Data Dependency. In short, if a calculation needs the result of a previous calculation, the instruction will not be dispatched until all its data items are available. In the meantime, other instructions which have received all the necessary data items will continue to be dispatched. For example:

<sup>17</sup>[https://en.wikipedia.org/wiki/Amdahl%27s\\_law](https://en.wikipedia.org/wiki/Amdahl%27s_law)

<sup>18</sup>[https://en.wikipedia.org/wiki/Instruction-level\\_parallelism](https://en.wikipedia.org/wiki/Instruction-level_parallelism)

Consider the following program:

```
1      1   a = x * y
2      2   b = x / y
3      3   c = x + y
4      4   d = x - y
5      5   e = a * b
6      6   f = c * d
```

In this example, operations 1, 2, 3 and 4 can all be performed in parallel. This is because the input operands *x* and *y* do not depend on other operations. Then operations 5 and 6 will be performed in parallel. While the instructions may be dispatched at the same time, they do not take the same amount of time to complete.

Addition and subtraction take fewer cycles to complete than multiplication and division. Because variable *b* is the result of the division (*x* / *y*), operation 5 will have to wait until “*b*” is available. In the meantime operation 6 can proceed before operation 5 because variables *c* and *d* are the result of addition and subtraction, which are much faster.

There are several ways to exploit ILP and speed up serial code which we will examine below. RISC++ is a hybrid system that uses a combination of techniques used by other theoretical and actual processors. RISC++ is designed to produce fast serial code with a simple instruction set that occupies a small foot print in memory.

The architecture defines two processor types; one for scalar instructions and data and the other for processing variable length vectors. The scalar part is optimized for ILP and the vector part is optimized for DLP.

## Dataflow Architecture

There has been much research for increasing ILP using [Dataflow Architecture](#)<sup>19</sup>. This family of designs has had a variety of proposed implementations, implemented in both hardware and software. The basic concept is to execute instructions in the *order that data arrives at the input* to the instruction rather than in the order specified by an Instruction Pointer. While I am not aware of any general-purpose computers using a strict data flow design at the ISA level, the concepts have been used in a variety of applications.

At the software level much research has been done in [Dataflow Programming](#)<sup>20</sup>. Some of this has involved new programming languages and new programming paradigms. But the marketplace continues to demand solutions that allow existing software to run faster on new hardware without being completely re-written. Current microprocessors use a data flow architecture under the covers as we see below.

---

<sup>19</sup>[https://en.wikipedia.org/wiki/Dataflow\\_architecture](https://en.wikipedia.org/wiki/Dataflow_architecture)

<sup>20</sup>[https://en.wikipedia.org/wiki/Dataflow\\_programming](https://en.wikipedia.org/wiki/Dataflow_programming)

## Very Long Instruction Word (VLIW)

The complexities and disadvantages of OoOE led Intel to invest a considerable amount of time and money in an ISA called [IA-64](#)<sup>21</sup> also known as Itanium. This processor family was designed to solve the problems created by the complex hardware needed to support X86 and OoOE. The job of reordering instructions to support ILP was moved from the hardware to the Compiler. Highly sophisticated compilers would examine the entire program's source code and determine which instructions could be executed in parallel. Then the parallel instructions would be grouped together in a [Very Long Instruction Word \(VLIW\)](#)<sup>22</sup>.

The IA-64 defined a 128-bit bundle of 3 instructions which were 41 bits each. The remaining 5 bits were used for routing instructions to various execution units. The 128-bit bundles could be linked together to create as many parallel instructions as the hardware could support but the maximum number of instructions that were dispatched per cycle was typically six.

Itanium had two major disadvantages that made it less of a marketing success than Intel had hoped. The first was that customers were reluctant to move away from the X86 with its huge inventory of compatible software. The second was that developing complex new compilers was very expensive. Still, while the Itanium never replaced X86 in popularity, it filled a niche market where it performed quite well.

## MAJC by Sun Microsystems

Another less well-known VLIW based Architecture was called [MAJC \(Microprocessor Architecture for Java Computing\)](#)<sup>23</sup>. This processor designed by Sun Microsystems was an attempt to leverage the fact that the JAVA Programming Language used a technique called [Just in Time Compilation \(JIT\)](#)<sup>24</sup> to dynamically convert Java Bytecode into machine language. JIT compilation is a powerful concept that continues to be a major part of [Microsoft .NET Framework](#)<sup>25</sup> and the programming languages that use it.

MAJC utilized a variable length VLIW instruction which were 1 to 4 instructions long. Each instruction was a fixed length of 32 bits. MAJC also used a single set of registers for integer and floating-point operations. This simplified the dispatching hardware and allowed for up to four floating-point or four integer instructions per cycle or any combination of the two. MAJC was only used internally at Sun, but the idea of using JIT to compile directly to a VLIW machine code was later used to emulate X86 as described below.

## Transmeta and Code Morphing

Just as Sun and the MAJC used JIT compilation to convert Java Byte Code into a VLIW based machine language, a company called Transmeta used [Code Morphing Software](#)<sup>26</sup> to translate X86

---

<sup>21</sup><https://en.wikipedia.org/wiki/IA-64>

<sup>22</sup>[https://en.wikipedia.org/wiki/Very\\_long\\_instruction\\_word](https://en.wikipedia.org/wiki/Very_long_instruction_word)

<sup>23</sup><https://en.wikipedia.org/wiki/MAJC>

<sup>24</sup>[https://en.wikipedia.org/wiki/Just-in-time\\_compilation](https://en.wikipedia.org/wiki/Just-in-time_compilation)

<sup>25</sup>[https://en.wikipedia.org/wiki/.NET\\_Framework](https://en.wikipedia.org/wiki/.NET_Framework)

<sup>26</sup>[https://en.wikipedia.org/wiki/Transmeta#Code\\_Morphing\\_Software](https://en.wikipedia.org/wiki/Transmeta#Code_Morphing_Software)

byte code into a proprietary VLIW. One of the advantages of Code Morphing was that new features of X86 could be added to the supported architecture features without changing the hardware. The simpler hardware also cost less, consumed less power and produced less heat. Larger blocks of binary code could be analyzed for translation than when decoding was done by hardware. But doing translation in software rather than hardware did affect the performance of Transmeta chips. As time went on, the advantages of lower cost, speed, power, and heat were overcome by advances in newer X86 chips designed by Intel and AMD. Transmeta eventually went out of business.

## Queued Operands

Another less known methodology for supporting ILP and speeding up serial code is using queued operands. In this methodology, register dependencies are eliminated by having instructions read input operands from the head of a Queue and write the results to the tail. This eliminates certain Data Hazards which prevent instructions to be executed in parallel. The use of queued operands greatly simplifies hardware design because the complexities of OoOE are eliminated. The instruction set is also very compact because the instructions can be one byte long with no data operands. For more information see the following paper produced in the Journal of Supercomputing:

[The QC-2 parallel Queue processor architecture](#)<sup>27</sup>

While RISC++ is not a Queue Processor in the strictest sense, it does use queued operands to facilitate dynamic allocation of registers for special queued instructions.

## Out of Order Execution (OoOE)

The most common way to speed up a serial instruction stream is called [Out of Order Execution \(OoOE\)](#)<sup>28</sup>. This method starts with a traditional [von Neumann architecture](#)<sup>29</sup> which uses an instruction pointer and branch instructions to define the sequence of instructions. Then as serial instructions are decoded they are placed in [Reservation Stations](#)<sup>30</sup> where they wait until all their input data are ready.

As the input data (source registers) become available, the instructions are sent to the appropriate parallel execution units (adders, multipliers, logical units, etc.) for execution. The instructions are often executed in a different order than the program sequence in memory. However, the results are written back to the registers in the original program order.

OoOE is used by most modern implementations of X86. It is also used in the IBM PowerPC and various other processors. The main advantage of OoOE is that it *preserves legacy binary code* while still improving single thread performance. The main disadvantages are that it *requires highly complex hardware* and that only a small window of binary code is optimized at a time.

---

<sup>27</sup><https://www.sciencedirect.com/science/article/abs/pii/S0743731507001554>

<sup>28</sup>[https://en.wikipedia.org/wiki/Out-of-order\\_execution](https://en.wikipedia.org/wiki/Out-of-order_execution)

<sup>29</sup>[https://en.wikipedia.org/wiki/Von\\_Neumann\\_architecture](https://en.wikipedia.org/wiki/Von_Neumann_architecture)

<sup>30</sup>[https://en.wikipedia.org/wiki/Reservation\\_station](https://en.wikipedia.org/wiki/Reservation_station)

## The Parable of the Bank

To illustrate this let us take the example of a line of people waiting patiently in queue at a bank. The queue splits into several smaller queues, one for each teller. As each person is serviced by a teller, the customers move on in a different order than when they came in. Some people come in with their checks signed and their deposit slips filled out. Others need to get out of line to sign the check and fill out the deposit slip.

Meanwhile other customers pass by the unprepared customers, get their work done and leave. Once the unprepared customer gets his check signed and deposit slip filled out he can rejoin the line. At the same time the various tellers take a different amount of time depending on the transaction. Some simply deposit a check with all the information ready. Some count a wad of cash. Others have to count pennies. Others sign people up for credit cards they really shouldn't have, but that's another story.

Some customers proceed quickly to the ATM and get their money right away. Still others use the drive through window and wait in line in their cars for a longer time. What does this have to do with computers? Let me explain.

## The Parable Explained.

RISC++ uses Out of Order Execution of instructions in the Typed Scalar Unit (TSU). However, the Fast Integer Unit and the Typed Vector Unit execute instructions in the order provided by the Instruction Pointer (IP).

Some of the Condition Bits, which are set as a result of comparing two Typed Scalar registers, are set out of order. If these bits are tested while they are still in the process of being set, the processor will stall. If the COMPARE instruction uses a register that is being set as the result of a DIVIDE instruction, it may wait a long time.

For this reason TSU conditional instructions, which are executed Out of Order, will wait in the Reservation Station for three data items; the condition bit, and the two input operands. If the condition bit is false, the instruction will be skipped and removed from the Reservation Station. If the CB is true, the instruction will be dispatched to the final stage of execution in the TSU.

The following instruction types are listed in the order of speed from slowest to fastest:

1. Slowest: Load from Memory when there is a cache miss.
2. Faster: Floating point Divide, Square Root, Sine, Cosine, etc.
3. Faster: Floating Point Multiply.
4. Faster: Floating Point Add, Subtract, Compare etc.
5. Fastest: Integer Add, Subtract, And, Or, Compare, etc.

Because instructions must wait in the reservation station until all the input operands are available, some may wait longer than others. The instructions are executed in a different order than that which

was defined by the Instruction Pointer and Branch instructions. OoOE facilitates [Instruction Level Parallelism \(ILP\)](#)<sup>31</sup>.

If the instruction is dispatched to the Fast Integer Unit (FIU) it executes very quickly, usually a single cycle. The FIU uses “*in order execution*”, each instruction must complete its operation before the other proceeds. This is like the customer who uses the ATM. He might need to stand in line to use it, but when he does he can finish his transaction quickly

If the instruction is dispatched to the Typed Vector Unit, it must wait in a queue until the two input operands are ready, but other instructions cannot bypass it. Since Vector instructions can take a long time to process, it could be a long wait. But like the FIU, the instructions proceed in the order in which they were dispatched. But once the Vector instruction is finally executed, it processes a large amount of data (256 bytes) in a single instruction cycle. In our bank analogy this could be the drive through window when there is only one teller.

---

<sup>31</sup>[https://en.wikipedia.org/wiki/Instruction-level\\_parallelism](https://en.wikipedia.org/wiki/Instruction-level_parallelism)

# Why Do We Need Another ISA?

This chapter is going to give a justification for introducing a new ISA at this point in time. Since this “hobby” has had me studying computer architectures over the a long period of time, I think it is good to review the way various computer architectures have evolved. Then I can show how RISC++ fits in the current state of technology. This chapter is going to have many links. I don’t expect people will want to look at them all. Some concepts that may be new to some readers will be explained better later in the document.

## RISC++ instruction set is simpler than X86

If you are a compiler writer, a developer of math libraries, someone who has written assembly level code for high performance vector processing (before GPGPUs and using SIMD), or if you are one of those “god like” people who have implemented the X86 Architecture in hardware, you will probably have no argument that X86 has become complex and cumbersome. But it also is an amazing piece of technology upon which the vast majority of computer software in the world today runs very well.

The reason why I believe there is room for another Instruction Set Architecture is because the X86 family of micro-processors have unnecessary complexity in their hardware due to the fact that they have evolved over the course of several decades. Since one of the requirements of the X86 hardware is to remain compatible with previous versions, this means that hardware must be more complex than if it only needed to support a new, streamlined, simple instruction set.

In the early days of programming, in both Assembler and C++, the programmer had much control over the binary code produced. But more control also meant that he or she had to spend much of time learning the intricacies of the hardware and less time in actually solving problems. Today there are all kinds of software tools and code libraries that insulate the programmer from the complexity of the underlying hardware. But the complexity is still there and the various “middle-ware” software must deal with it.

If the design of the RISC++ instruction set and hardware is sound, (and I believe it is), programmers will see very little difference in how they write code. In fact they probably won’t even know if the code will be run on a RISC++ chip, since many of them will be in cloud based servers where only the bravest geeks dare go. But the compiler will generate more streamlined and efficient code.

But wait, you might say, memory is cheap and fast today, why bother economizing? The answer is that on-chip caches are still premium hardware real estate and any economies will yield great rewards in processor performance. Not only does the generated code take up less memory. It also has much shorter path lengths so it runs faster.

I can hear the next objection. In the last chapter I argued the *CISC* not *RISC* ISAs have the smaller memory foot print. This is true to a point. The variable instruction length and memory to memory

instructions do take up less space in the Instruction Cache. But I think RISC++ binary code will use less memory because a *single code image* supports many data types.

This is especially important when [Operator Overloading](#)<sup>32</sup> is used to define various data types or combination of data types. When HLL code with Operator Overloading is compiled to most ISAs, like the X86, there must be many *type specific* copies of the binary code.

With RISC++, only the Load and Store instructions need to be type specific. All the Typed Scalar arithmetic instructions are generic in regards to data type, and thus only one binary code image is needed. Since typed registers can be loaded before calling a subroutine, then passed on the stack, the subroutine itself does not need any type specific arithmetic instructions.

Take time to scan the [x86 and AMD64 instruction reference](#)<sup>33</sup>. What do you notice? How many different versions of ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPARE, CONVERT, instructions are there? In X86 (as with most architectures) there is a unique instruction for each data type.

Once you start putting multiple data types inside 64 and 128 bit registers, and then allow combinations of types in a single instruction, the number of instructions increases exponentially. Imagine how complex the compiler has to be and how many different code libraries are needed to generate code for various combinations of data types, especially with “operator overloading”. (I assuming some knowledge of C++ programming here).

The X86-64 instruction set has a very large number of instructions. A great number of these support various data types and the combinations of types. I am referring to data types such as 8, 16, 32 and 64 byte integer (signed and unsigned) as well as 32 and 64 bit Floating Point. These types are so common that no one seems to think how many instructions it takes the manipulate them, especially when there are both scalar and vector instructions that use them.

When high level languages allow the mixing of data types in an operation it seems so easy to the programmer. But under the covers the compiler has to determine which instructions to use out of a vast array of choices. In the past, not all chips supported the full instruction set, so that made the job of the compiler even harder as it needed to test bits to determine which features were available on the particular chip is was running on. Thank goodness the X86-64 instruction set seems to have stabilized. But it is still very complex.

## X86 Architecture Overview

In this document, the term [X86](#)<sup>34</sup> includes the ISA that is implemented on many families of software compatible chips designed by Intel, AMD, and other companies. Looking at the linked Wikipedia article will show what a large number of versions of X86 are out there and how many companies manufacture them. X86 includes both 32 and 64-bit scalar instruction sets and many variants of vector (SIMD) instruction-sets. It also includes a complex variety of options to handle real and virtual memory addressing.

---

<sup>32</sup>[https://en.wikipedia.org/wiki/Operator\\_overloading](https://en.wikipedia.org/wiki/Operator_overloading)

<sup>33</sup><https://www.felixcloutier.com/x86/index.html>

<sup>34</sup><https://en.wikipedia.org/wiki/X86>



Over time, the scalar instructions of X86 have evolved from 8 to 64-bit computers with considerable duplication of function. New innovations and higher clock speeds continue to squeeze more and more performance out of the chips. Deep pipelines, Out of Order Execution, replication of resources, and other techniques all work together to produce higher and higher performance of an aging architecture based on a serial instruction stream. However, this has not come without a cost. A significant amount of resources are dedicated to supporting the complex instruction set, rather than being devoted to speed and functionality.

But the real complexity was in the compilers that tried to keep up with an ever evolving set of instructions which were not implemented on every chip. For this reason the SSE instructions were not implemented in a transparent way to the programmer. Instead the programmer had to drop down into assembler if he or she really wanted to get the most performance possible. Some C++ compilers included intrinsic data types and instructions which were basically allowing direct access to the hardware without resorting to Assembler programming. But the code was very machine specific and could not be easily ported from one processor type to another.

Reading the following two articles will give a good overview of the X86 ISA and how it evolved over time:

1. [IA-32<sup>35</sup>](#): The 32 bit version of the X86 architecture which lasted many years.
2. [X86-64<sup>36</sup>](#): The 64 bit version which almost all new processors implement.

## X86 Single Instruction Multiple Data (SIMD)

Long before graphics processors were around software had to process large blocks of data which represented the graphics to be displayed on the screen. The CPU had to fill the screen buffer very quickly especially as screen resolutions increased and games required animations in real time.

The 16/32 bit processors of that time were no match for the demands of gaming computers. Over time, and as memory prices dropped, graphics coprocessors emerged and eventually evolved to the processing powerhouses they are today. Now we have high resolution live video on our phones and we seem to take it for granted. But it was not always this way.

At first, the problem of processing large parallel data sets was done in the CPU using instructions that processed several bytes of parallel data in a single register. If you have not yet done so please read the Wikipedia article on SIMD linked above. Then, if you wish, follow the links of each of the variations on SIMD which evolved over time. Below I will very briefly summarize the main points.

### MMX: Introduced in January 1997 by Intel.

- Used to speed up the processing of parallel *integer* data for graphics.
- Instructions used 64 bit floating point registers to hold parallel integer data.

---

<sup>35</sup><https://en.wikipedia.org/wiki/IA-32>

<sup>36</sup><https://en.wikipedia.org/wiki/X86-64>

- Internal 80 bit Floating Point<sup>37</sup> registers use NAN<sup>38</sup> encoding to flag integer data.
- Integer data types: 8 x 8-bit, 4 x 16-bit, 2 x 32-bit, 1 x 64-bit.
- Pro: Using Floating Point registers allowed Operating System to save and restore registers during context switching without modification.
- Con: Sharing Regs between Floating Point and graphics software slowed both down.

## 3DNow!: SIMD ISA by Advanced Micro Devices (AMD) in 1998.

- Like MMX, it used Floating Point Registers with the same pros and cons.
- Added Vector Floating Point using two side by side 32-bit floats in a 64-bit register.
- Add, Subtract, Multiply Vector float. Convert to/from integer and float various lengths.
- Reciprocal approximation followed by Multiply instead of Divide.
- Square root and Reciprocal Square root approximation.
- Average of various integer types (8, 16, 32, signed unsigned) horizontally in 8-byte register.
- Special instructions to speed up context switching and cache management.
- Not popular with software developers. Support dropped in August 2010

In 1999 Intel introduced a whole new set of instructions to handle SIMD called Streaming SIMD Extensions (SSE) as part of the Pentium III<sup>39</sup> processor. The Pentium III was a 32 bit processor and did not have many of the features now found in X86-64 processors.

Also, in the meantime, Intel was working on the Itanium Architecture<sup>40</sup> (also known as IA-64) as its new entry into 64 bit computing. We will talk about the pros and cons of this architecture later. For now let it suffice to say that it was not at all compatible with the 32 bit X86.

AMD saw this and got to work on its own version of a 64 bit Architecture which was backwards compatible with the 32 bit version This was rightly called AMD-64. AMD broke new ground but it wasn't long before Intel realized that the AMD-64 was the future and began making its own versions Now it is the dominant architecture world wide.

The following is a summary of the evolution of SSE:

## SSE Extension of X86-32 SIMD introduced in 1999.

- Continued to use MMX instructions for integer operations mapped to Floating Point registers.
- Added 8 new 128 bit registers for 4 x 32 bit Floating Point Data.
- Introduced new scalar and "packed" (vector) operations.
- Scalar/Vector: Add, Subtract, Multiply, Divide, Reciprocal, Square Root, Reciprocal Square Root, Minimum, Maximum.
- Various data shuffling, conversion, compare, bit manipulation and memory access instructions.

<sup>37</sup>[https://en.wikipedia.org/wiki/Extended\\_precision](https://en.wikipedia.org/wiki/Extended_precision)

<sup>38</sup><https://en.wikipedia.org/wiki/NaN>

<sup>39</sup>[https://en.wikipedia.org/wiki/Pentium\\_III](https://en.wikipedia.org/wiki/Pentium_III)

<sup>40</sup><https://en.wikipedia.org/wiki/Itanium>

## **SSE2 Extension of 32 bit Pentium 4 introduced in 2000.**

- SSE2 added 144 new instructions to SSE, which has 70 instructions.
- Uses 4 x 32 and 2 x 64 bit floating point data in 128 bit registers.
- Included 16 x 8, 8 x 16, 4 x 32, 2 x 64 bit integers.
- Same basic scalar and vector instructions as SSE with new data types.
- Large number of instructions to accommodate various data types and combinations of types.
- Smaller Floating Point data causes round off errors.
- Here is the full [X86-64 Instruction Listings for SSE2](#)<sup>41</sup>

## **SSE3 Intel introduced SSE3 in early 2004**

- SSE3 contains 13 new instructions over SSE2
- Added more horizontal instructions which added and subtracted data across the 128 bit registers.
- Instructions to support array of structures.
- MONITOR, MWAIT - These optimize multithreaded applications, giving processors with [Hyper-threading](#)<sup>42</sup> better performance.
- NOTE: Hyperthreading is not used in RISC++. Instead the architecture tries to improve performance of serial threads, as will be explained in the next chapter.

## **SSSE3 Supplemental Streaming SIMD Extensions:**

- Introduced with Intel processors based on the [Core micro-architecture](#)<sup>43</sup> in June 2006
- SSSE3 contains 16 new discrete instructions.
- Each instruction can act on 64-bit MMX or 128-bit XMM registers.
- Twelve instructions that perform horizontal ADD or SUBTRACT.
- Six instructions that evaluate absolute values.
- Two instructions that perform “multiply and add” operations.
- Two instructions that accelerate packed-integer multiply operations.
- Two instructions that perform a byte-wise, in-place shuffle.
- Six instructions that negate packed integers.
- Two instructions that align data from two operands.

---

<sup>41</sup>[https://en.wikipedia.org/wiki/X86\\_instruction\\_listings#SSE2\\_instructions](https://en.wikipedia.org/wiki/X86_instruction_listings#SSE2_instructions)

<sup>42</sup><https://en.wikipedia.org/wiki/Hyper-threading>

<sup>43</sup>[https://en.wikipedia.org/wiki/Intel\\_Core\\_\(microarchitecture\)](https://en.wikipedia.org/wiki/Intel_Core_(microarchitecture))

## SSE4 Announced on September 27, 2006

- SSE4.1 consists of 54 instructions.
- SSE4.2, a second subset consisting of the 7 remaining instructions
- Multiplication of different size integers in vector (packed).
- Dot Product of arrays.
- Conditional copying based on mask bits.
- Packed minimum/maximum for different integer operand types
- Round values in a floating-point register to integers, using one of four rounding modes specified by an immediate operand

## SSE5 Proposed by AMD on 2007

- NOTE: the linked article is confusing about what instructions were actually implemented and which were dropped.
- The proposal is included as reference to the kind of ideas that were being discussed at the time.
- AMD proposed 256 **bit** SIMD, RISC++ has 256 **byte** variable length vectors.

Various compilers sought to use normal C++ “for loops” to generate code directed at various SIMD instruction sets. This is called [Automatic vectorization](#)<sup>44</sup>. Today the [Intel C++ Compiler](#)<sup>45</sup> and the [GNU Compiler Collection](#)<sup>46</sup> seem to have conquered the very complex issues of Automatic Vectorization.

It is my belief, however, that the way RISC++ handles scalar and vector data facilitates Automatic Vectorization seamlessly. This is because the instructions already work together to provide easy compilations of “for loops”. But I will need experts in compiler design to refute or validate that presumption.

## A brief History of RISC++

When I first started designing RISC++ there were two essential ideas which have never changed. The first was dynamically typed registers. The second was conditional instructions. I could have published back then and people would have said “cool, but why?”. Who cares what the hardware is doing or what the Assembler Language and machine code looks like, as long as my High Level Language (HLL) code works.

As time when on this “hobby” was a welcome distraction from my “real job” as a pastor. I enjoyed reading about the evolution of the X86, and reading up on other architectures such as IBM Power PC, DEC Alpha, Intel Itanium, and others.

---

<sup>44</sup>[https://en.wikipedia.org/wiki/Automatic\\_vectorization](https://en.wikipedia.org/wiki/Automatic_vectorization)

<sup>45</sup>[https://en.wikipedia.org/wiki/Intel\\_C%2B%2B\\_Compiler](https://en.wikipedia.org/wiki/Intel_C%2B%2B_Compiler)

<sup>46</sup>[https://en.wikipedia.org/wiki/GNU\\_Compiler\\_Collection](https://en.wikipedia.org/wiki/GNU_Compiler_Collection)

But I was always looking at the ISA and trying to think through how to go from C++ to the machine code architecture, and then how to implement that machine code in real hardware. As I have hopefully demonstrated in this chapter, the greatest challenge is in how to make HLL “*for loops*” translate into fast Vector instructions.

A preliminary version of RISC++ 1.0 was published on Arstechnica.com and several people made comments. At the time, I was trying to incorporate Scalar and Vector (SIMD) instructions into a single thread by using a unified set of 128-bit registers. The goal was to simplify the hardware by having a single execution unit, a single instruction set, and a single register set, for all the operations in a thread of operation.

Many of the comments were about the rare use of 128-bit scalar data, and the wasted space in the registers when 8, 16, 32 or 64-bit scalar data was needed. 128-bit registers are fine for SIMD instructions but are inefficient for scalar. Another major complaint was the use of dynamic register saving when using 128-bit registers. I also did not explain the “why” of dynamic typing very well.

So, I archived version 1.0 and started working on version 2.0. That version had two completely different processor types, one for scalar and another for vector operations. That configuration was much like the use of a general purpose Central Processing Unit (CPU) for scalar operations and General Purpose Computing on Graphics Processing Units (GPGPU) for vector operations. These two processor types are used extensively in various CPU and GPU chips manufactured by Intel, AMD and Nvidia.

The main difference between RISC++ 2.0 and Graphics Based hardware was that the RISC++ ISA supported more IEEE 754 Data Types and exceptions. The Vectors were massive as they are in GPGPU applications. Also the RISC++ vector hardware had no graphics specific hardware. The scalar architecture was about what it is today. (TSRs and Conditions bits have been around since the early '90s). But there were all kinds of problems inherent in having a [Heterogeneous System Architecture](https://en.wikipedia.org/wiki/Heterogeneous_System_Architecture)<sup>47</sup>.

The main problem was that this approach was moving away from the primary goal of RISC++, to produce a simple binary code generated from a C++ like compiler. The CPU/GPU configuration requires an Application Programming Interface (API) like [CUDA](https://en.wikipedia.org/wiki/CUDA)<sup>48</sup> to utilize the GPU for vector operations. CUDA helps the programmer by handling the issues involved with a specific hardware configuration, synchronization of threads, etc.

That configuration was originally designed for Graphics Processing and later adapted for general purpose vector processing. Such an approach has proven to be very successful for many reasons, not the least of which is the ubiquity of X86 CPUs and graphics processor cards made popular by the gaming industry. But this hardware configuration requires a hardware specific programming paradigm.

RISC++ is not intended to replace existing hardware and software, but to provide a niche solution for “*scientific processing*”. Scientific processing makes heavy use of the 64-bit binary floating-point

<sup>47</sup>[https://en.wikipedia.org/wiki/Heterogeneous\\_System\\_Architecture](https://en.wikipedia.org/wiki/Heterogeneous_System_Architecture)

<sup>48</sup><https://en.wikipedia.org/wiki/CUDA>

numerical format for both scalar and vector operations. Therefore, the hardware has been optimized for this format, while also supporting other formats.

RISC++ Version 3.0 (now simply RISC++) defines a configuration with a Typed Scalar Unit (TSU), which supports Instruction-level parallelism (ILP) and a Vector Processing Unit (VPU), which implements Data Parallelism. Both units are accessed by a *single thread* of instructions. RISC++ also defines an Instruction Control Unit (ICU), a Load/Store Unit (LSU) and a Fast Integer Unit (FIU). We have introduced these units and will examine them in more detail later in the document.

## Intel's oneAPI and Data Parallel C++

Intel has been working hard on the problem of supporting various kinds of processors through a single Application Programming Interface (API). Their solution, which is quite recently announced, is called the [Intel® oneAPI Toolkits](#)<sup>49</sup>. This API supports a new Intel C++ compiler called [Intel® oneAPI Data Parallel C++](#)<sup>50</sup>. Together, the API and the Compiler, take “normal” C++ coding practices and produce machine code which can run on the following processor types:

1. [Central processing unit \(CPU\)](#)<sup>51</sup>
2. [Graphics processing unit \(GPU\)](#)<sup>52</sup>
3. [Field-programmable gate array](#)<sup>53</sup>
4. And various other specialized accelerators.

I have not yet had time to fully study Data Parallel C++ and adapt RISC++ to fit. As I said, this whole project has had me chasing after evolving technology and never getting published. I am hoping that the way RISC++ handles vectors already supports the Intel compiler constructs. If not, it is still the goal of RISC++ to support legacy C++ programs without requiring a new programming paradigm.

Tools like Data Parallel C++ insulate the programmer from the complexity of the X86 instruction set and hardware, but that complexity is still there “under the covers”.

In RISC++, the combination of *typed registers*, *typed variable length vectors* and *conditional instructions* facilitate compact code to be fetched from the Instruction Cache and to be executed with a minimal number of branches.

Also, while the X86 hardware uses 128 bit registers to facilitate Vector operations, each RISC++ vector instruction operates on 16 x 128 bits (256 bytes) in every cycle. A single RISC++ vector instruction like ADD VECTOR, MULTIPLY VECTOR, etc. replaces complex SIMD used in the X86 ISA. The RISC++ Assembler code also has a close relationship to the C++ source code and can be read intuitively by human beings.

---

<sup>49</sup><https://software.intel.com/en-us/oneapi>

<sup>50</sup><https://software.intel.com/en-us/oneapi/dpc-compiler>

<sup>51</sup>[https://en.wikipedia.org/wiki/Central\\_processing\\_unit](https://en.wikipedia.org/wiki/Central_processing_unit)

<sup>52</sup>[https://en.wikipedia.org/wiki/Graphics\\_processing\\_unit](https://en.wikipedia.org/wiki/Graphics_processing_unit)

<sup>53</sup>[https://en.wikipedia.org/wiki/Field-programmable\\_gate\\_array](https://en.wikipedia.org/wiki/Field-programmable_gate_array)