Cellia
A Homebrew ISA

Ashton Scott Snapp

August 7, 2020

Contents

Preface

This book, in whatever form you find it in (digital or physical), is meant to document my work-in-progress instruction set architecture, which I have dubbed Cellia.

Initially, I had only planned to go over the features of the architecture along with how to program for it and design around it. However, after encountering the YouTube Channel grokThis, their MicroCISC (uCISC) architecture, and having a gist chat with the man behind it, it became apparent that that (what an odd construction) simply would not be enough. I need to go over why I'm doing this, what my design goals are, and going through each decision I have made and will have to make in the future to get this implemented.

So, that is what I will be doing in this book. First the story, then the design decisions, then getting into code and hardware design. Finally, I will talk about the future, from implementing it to extending the architecture. Of course, you likely already gathered that order from looking at the table of contents.

Part I

The Story

Chapter 1

Beginnings

It all started the same way a lot of other homebrew computing projects started - David Murray. The 8-Bit Guy. If you did not know, back in 2018 David created a post on his website detailing his dream computer. It listed a few requirements - off-the-shelf components if possible, a 6502 or compatible processor, at least 128K of RAM, stuff like this. This post kicked off projects like the C256 Foenix. It also piqued my interest as well, and I started looking into how I could make a computer using the 65816.

As it turns out however, there is a reason that only three systems ever used the 65816 processor (or a variant) as its primary processor (those being the Acorn Communicator, Apple IIGS, and the SNES) - it's a pain to work with. Firstly, the address and data busses are multiplexed. Multiplexing means that certain pins on the chip can be one of two different signals. In this case, the 8-bit external data bus also acts as the most significant 8-bits of the address bus. This can be worked around using some hardware, but that's not the only problem. Another issue is trying to create a memory map. One thing that the 65816 does for backwards compatibility is placing the interrupt vectors in bank 0, with a bank being 64K large, and the interrupt handlers these vectors point to also have to be in bank 0. At first this doesn't seem like a problem, just put a rom chip in bank 0. That's until you realize the stack also must be located in bank 0. This means that either you need to complicate your memory map (and by extension your address decode logic) to have both ROM and RAM in bank 0, or have an external piece of hardware copy over 64K of the ROM over to the RAM in bank 0, which adds unneeded complexity.

So, the first thing that comes to your mind is to go look for an alternative.

And so I did, onto Mouser.com I went! But, to my horror, pretty much all of the 16-bit alternatives that were in stock had some of the same issues, and even their own. First up on the chopping block, the Zilog Z8000, a 16-bit (incompatible) equivalent of their popular Z80. It also wasn't very popular back in the 80s, at least for home computers, only really being used for around 10 systems. Okay it was technically 11 but one of them was cancelled. The original Z8000 had two variants: one had a 23-bit bus that could access 8M and the other could only use 64K (which kinda defeats one of the purposes of using a 16-bit CPU). Now, with the Z8000 now being called the Z1600, the variants are known as the Z1601 (23-bit address) and the Z1602 (16-bit address). Sadly, only the Z1602 is listed on Mouser. In addition, the address and data bus pins are again multiplexed, but even worse this time! As both the address and data busses are 16-bits long on this variant, they take up the exact same pins! WHY

In addition, Renesas Electronics still creates a CMOS version of the 8088, however it still has multiplexed pins. Why did everyone in the 70s and 80s think this was a good idea?

Eventually, I gave up on using an off-the-shelf 16-bit CPU and went to the trusty 6502. However, I still wanted to create 16-bit homebrew computers. The solution to that want? Cellia.

Chapter 2

Design Goals

If I'm going to do this, I want to do this right. But, right means different things to different people so it would be best to put down my design goals - what it would mean to do this right in my eyes.

First,

designing a computer to use a Cellia processor should not make the process of designing said computer more difficult.

Second,

I/O devices must not lower the amount of memory (volatile or otherwise) you can put in your computer designs.

Third,

Cellia processors should be easy to replace, either if they break or if a better version comes out. What can I say, I'm a fan of right to repair.

Fourth,

Other people should be able to create Cellia processors via a public and open FPGA core. The architecture must be able to continue without me if

necessary. The most likely open hardware license will be the CERN OHL v2, specifically the permissive variant.

Part II

Decisions

Chapter 3

CISC or RISC

To RISC or not to RISC? When it comes to designing processor architectures, there are two general approaches, known as CISC (complex instruction set computing) and RISC (reduced instruction set computing). I'll try to explain their differences to the best of my abilities here before telling you what I chose.

CISC was the first one to appear. This is your x86, 65xx, 68k, and uCISC. In a CISC architecture, a single instruction will do several operations at once. For example, an add instruction might perform one or two memory loads, perform the addition, then perform a memory store. While this takes up less space on the ROM or whatever media the code is stored on, these instructions take more time as they have to do multiple things.

RISC came after CISC. This is your ARM, SPARC, MIPS, and PowerPC. RISC architectures have instructions that do only one thing. Loading a value into a register is one instruction, addition is another, then storing the result back into memory is another as well. This does take up more space on whatever the code is stored on, but it is much faster. Combined with some pipelining trickery, and a RISC instruction often takes only one or two clock cycles to complete.

And now, the moment you've all been waiting for (unless you knew before reading this), Cellia will be a.... (drum-roll) RISC architecture! ... I'm going to be honest, I don't know how I'm going to finish this paragraph. Let's just go to the next chapter.

Chapter 4

Busses

All processors have at least two busses: a data bus and an address bus. These busses have a size, and the data bus can differ internally and externally. For example, the 8088 has an 8-bit external data bus, 16-bit internal bus, and 20-bit address bus. Some processors also have a third bus, called a port bus, which is where I/O devices are addressed by the processor. This also has a size. In the case of the 8088, that size is 16-bits.

When you don't have a port bus, your only way of addressing I/O devices is over the same address bus you use for memory. This is referred to as Memory Mapped I/O, as I/O is treated the same as memory. This is present on processors like the 6502, but can really be used on any processor even if they have a port bus. If a processor does have a port bus, that processor uses what is called Port Mapped I/O.

Why do some processors rely on MMIO while others havve PMIO? Well, most of the processors that have PMIO can only access it using specific instructions and specific registers, and often ports cannot be used for the same things that addresses can. For example, x86 has specific in and out instructions for accessing the port bus. When using these instructions, only the A register(s) can be used. However, neither approach is considered superior and both are widely used.

For Cellia, I have decided to go with PMIO. The main issue with PMIO, the restrictive instructions and registers, can be easily circumnavigated with the architecture design.

For the size of the busses, the address bus will be 24-bits long, allowing a Cellia processor to access 16 mebibytes (16*(1024*1024)) of RAM and ROM, while the port bus will be 16-bits long. The internal data bus will always be

16-bits long, though I am thinking of having variants with 16-bit and 8-bit external data busses, like how there's an 8086 and an 8088.

Part III

Programming

Chapter 5

Registers

Cellia is currently planned to have eightish general purpose registers. All of these are "integer" registers, meaning they are whole numbers. They are as follows: iA (which can be split into iAH and iAL), iB (which can be split into iBH and iBL), iX, and iY. iA, iB, iX, and iY are all 16-bit, while iAH, iAL, iBH, and iBL are 8-bit. Specifically, the one labelled H is the high byte of their respective 16-bit register while the one labelled L is the low byte. These will be easy to implement as, from what I know, Verilog allows you to access slices of registers, like how Rust allows you to access slices of strings.

After the general purpose registers we have the stack pointer. The stack takes up an entire bank (65,536 bytes) of RAM, where the exact bank is chosen by loading the byte located at hex FFFFF0 into a hidden "Stack Bank" register. As such, the stack pointer is 16-bits long.

Then we have the instruction pointer, which points to the start of the next instruction to be executed. As it would need to access the entirety of the memory map, it is 24-bits long.

Finally, there is the FLAGS register (specifically the iFLAGS register, gotta futureproof!). As its name suggests, it stores a bunch of flags that correspond to the status of the processor. It is 16-bits long and all 16-bits are used, as follows:

Bit	Flag	Name
0	Z	Zero Flag
1	C	Carry Flag
2	H	Half-Carry Flag
3	O	Overflow Flag
4	N	Negative Flag
5	S	Sign Flag
6	B	Borrow Flag
7	De	Decimal Flag
8	IL0	Interrupt Level Bit 0
9	IL1	Interrupt Level Bit 1
10	IL2	Interrupt Level Bit 2
11	I	Interrupt Disable Flag
12	E	Stack Empty Flag
13	F	Stack Full Flag
14	Di	Direction Flag
15	P	Parity Flag

And now, to explain the table you just saw. First, the Zero Flag. When Z is set (i.e it is 1), then the result of the last instruction was zero. Simple.

The Carry and Half-Carry Flags are a bit more complicated. These are used in certain arithmetic instructions. The carry flag indicates if a carry was generated out of the last instruction. Similarly for the half-carry flag, except it indicates if a carry was generated out of the last four bits.

The Overflow flag indicates an arithmetic overflow.

The Negative and Sign flags are quite similar. The main difference is that the Negative flag is used in unsigned situations while the Sign flag is used in signed situations. And by situations I mean branch instructions.

The Borrow flag is used in non-carry subtractions as a borrow. Set it before doing non-carry subtractions.

The Decimal flag switches the processor between normal mode and Binary Coded Decimal mode, where every four bits indicates one of the 10 decimal digits. More information can be found somewhere on the internet.

The ILx bits are used to represent the interrupt level. If the processor gets a maskable interrupt request and it isn't disabled, these bits will show the lowest interrupt level because for some reason lower = higher priority.

The Stack Empty and Stack Full flags are self-explanatory.

The Direction flag indicates whether the address should be incremented or decremented when reading multi-byte values.

The Parity flag is set or cleared depending on the parity of whatever the processor just read in, if the Data Parity pin is connected to memory.

Chapter 6

Addressing

While there is only one instruction that can address memory directly, there are multiple ways to do so.

The most straight-forward way is called Absolute Addressing. With absolute addressing, the instruction takes in the entire 24-bit (address bus) or 16-bit (port bus) address as an argument. As an argument for an instruction, these are represented as a for Absolute Memory Addressing and p for Absolute Port Addressing.

When addressing memory, you have two more options: Zero Bank addressing, which assumes the bank is zero, and Zero Page Addressing, which assumes that the bank and page (256 bytes) are zero. These are represented as zb (zero bank) and zp (zero page) as instruction arguments.

Finally, there is also relative addressing, where you can access memory relative to either the instruction pointer (ipr) or the stack pointer (spr).

Another thing: the iX and iY registers can be used as indexes on absolute, zero bank, and zero page addressing modes. This is represented with a , iX or , iY after the arguments on the instruction. In the cases of absolute and zero bank, their full 16-bit contents are used. However, since the zero page can only be addressed with 8-bits, only half of these registers are used to index zp.

Chapter 7

Instructions

Currently, the Cellia ISA has 91 operation codes (opcodes) which represent 43 instructions. Each opcode takes anywhere from zero to two arguments. These arguments can be any of the addressing modes listed in the previous chapter, along with immediate values and registers, represented as # and r respectively. There are also implied arguments, like the stack or the index registers.

7.1 NOP

The most basic instruction, it does nothing sucessfully. Represented in machine code as hex 00.

7.2 MOV

The only instruction that can mess with memory directly, and the one with the most opcodes, this is the move instruction. As you might guess, this instruction moves stuff between memory and registers. In total it takes up 30 opcodes, from hex 01 to hex 0E, and from hex 10 to hex 1F. I'm not entirely sure how I'm supposed to do a list in LaTeX, so I'll just do individual sentences.

Hex 01 is used to move an immediate value into a register, represented as MOV #, r.

Hex 02 is used to move a value from an absolute memory address into a register, represented as MOV a, r.

Hex 03 is used to move a value from a zero bank memory address into a register, represented as MOV zb, r.

Hex 04 is used to move a value from a zero page memory address into a register, represented as MOV zp, r.

Hex 05 is used to move a value from an absolute port address into a register, represented as MOV p, r.

Hex 06 is used to move a value from an instruction pointer relative address into a register, represented as MOV ipr, r.

Hex 07 is used to move a value from a stack pointer relative address into a register, represented as MOV spr, r.

Hex 08 is used to move a value from register into another register, represented as MOV r, r. The first register is the source while the second register is the destination.

Hex 09 is used to move a value from a register to an absolute memory address, represented as MOV r, a.

Hex 0A is used to move a value from a register to a zero bank memory address, represented as MOV r, zb.

Hex 0B is used to move a value from a register to a zero page memory address, represented as MOV r, zp.

Hex 0C is used to move a value from a register to an absolute port address, represented as MOV r, p.

Hex 0D is used to move a value from a register to an instruction pointer relative address, represented as MOV r, ipr.

Hex 0E is used to move a value from a register to a stack pointer relative address, represented as MOV r, spr.

Hex 10 through 13 are the same as hex 02 through 05, but indexed with the iX register, represented as MOV ?, r, iX.

Hex 14 through 17 are the same as hex 02 through 05, but indexed with the iY register, represented as MOV ?, r, iY.

Hex 18 through 1B are the same as hex 09 through 0C, but indexed with the iX register, represented as MOV r, ?, iX.

Hex 1C through 1F are the same as hex 09 through 0C, but indexed with the iY register, represented as MOV r, ?, iY.

7.3 HACF

This instruction, Halt and Catch Fire, does what it says on the tin. It essentially prevents the computer from running until a reset occurs. Represented as opcode hex 0F.

7.4 PUSH

This instruction takes a value from a register and pushes it onto the stack, represented as PUSH r. Occupies opcode hex 20.

7.5 PSHF

This instruction takes the current status of the iFLAGS register and pushes it onto the stack. Occupies opcode hex 21.

7.6 POP

Pops a value off the top of the stack and puts it in a register, represented as POP r. Occupies opcode hex 22.

7.7 POPF

Pops a value off the top of the stack and puts it in the iFLAGS register. Occupies opcode hex 23.

7.8 SET

Given a bit mask and a register, sets the masked bit in that register. Represented as SET bm, r. Occupies opcode hex 24.

7.9 SETF

Given a bit mask, sets the masked bit in the iFLAGS register. Represented as SETF bm. Occupies opcode hex 25.

7.10 CLR

Given a bit mask and a register, clears the masked bit in that register. Represented as CLR bm, r. Occupies opcode hex 26.

7.11 CLRF

Given a bit mask, clears the masked bit in the iFLAGS register. Represented as CLRF bm. Occupies ocpode hex 27.

7.12 ADDI

The first ALU operation. In this case, it is add without carry. The first argument is added to the second argument. The first argument can be an integer register or an immediate value, while the second argument is always an integer register. Represented as ADDI ir, ir or as ADDI #, ir. The two register opcode occupies hex 28 while the immediate and register opcode occupies hex 29.

7.13 ADCI

The same as ADDI, but now using the Carry Flag. So, add with carry. Has the same set of arguments as ADDI, so just take its syntax and replace ADDI with ADCI. Occupies hex 2A (ADCI ir, ir) and 2B (ADCI #, ir).

7.14 SUBI

Same as ADDI, but with subtraction. I think you know the drill by now. This instruction occupies the opcodes hex 2C and 2D. Make sure to set the borrow flag before doing a no-carry subtraction.

7.15 SBCI

Same as ADCI, but with subtraction. Uses the carry flag in a similar manner as the 65xx uses the carry flag for its SBC instruction. Occupies the opcodes

hex 2E and 2F.

7.16 AND

Does a logical AND on the two arguments. Takes the same argument set as the arithmetic instructions ADDI/ADCI/SUBI/SBCI. Occupies the opcodes hex 30 and 31.

7.17 OR

Does a logical OR on the two arguments. Occupies the opcodes hex 32 and 33.

7.18 XOR

Does a logical eXclusive OR on the two arguments. Occupies the opcodes hex 34 and 35.

7.19 NOT

Takes a register and flips all its bits. (NOT r) Occupies opcode hex 36.

7.20 SHL

Takes a register and shifts its bits left. Occupies opcode hex 37.

7.21 SHR

Takes a register and shifts its bits right. Occupies opcode hex 38.

7.22 ROL

Takes a register and rotates its bits left. Occupies opcode hex 39.

7.23 ROR

Takes a register and rotates its bits right. Occupies opcode hex 3A.

7.24 PCNT

Takes a register, counts the number of set bits, then stores the count in another register (PCNT r, r). Occupies opcode hex 3B.

7.25 VCNT

Similar to PCNT, but counts the number of cleared bits (VCNT r, r). Occupies opcode hex 3C.

7.26 JUMP

Redirects code execution to the given absolute address (JUMP a) or instruction pointer relative address (JUMP ipr). Occupies opcodes hex 3D and 3E.

7.27 CALL

Calls a subroutine located at the given absolute address (CALL a). Occupies opcode hex 3F. Pushes a return address onto the stack, which is the first byte after the end of the address.

7.28 CMPI

Compares two integer registers (CMPI ir, ir) or an immediate value and an integer register (CMPI #, ir). Really it just does a subtraction and drops the result. Occupies opcodes hex 40 and 41.

7.29 TEST

Given a bit mask and a register, checks if the masked bit is set (TEST bm, r). Sets the zero flag if it is set, clears it otherwise. Occupies opcode hex 42.

7.30 TSTF

Same as TEST, but checks the masked bit on the iFLAGS register. Occupies opcode hex 43.

7.31 Branch Instructions

All branch instructions have two opcodes: the even one jumps to an absolute address if the condition is true while the odd one jumps to an instruction pointer relative address if the condition is true. I think this time I'm just gonna use a table.

a	ipr	Mnemonic	Condition
44	45	BEQU	Z is set
46	47	BNEQ	Z is clear
48	49	BLTU	N is set
4A	4B	BLTS	S is set
4C	4D	BGTU	N is clear
4E	4F	BGTS	S is clear
50	51	BLEU	N or Z is set
52	53	BLES	S or Z is set
54	55	BGEU	N is clear or Z is set
56	57	BGES	S is clear or Z is set

7.32 RETS

Used to return from a subroutine. Pops the return address off of the stack and jumps to it. Occupies opcode 58.

7.33 BRK

Causes a software interrupt. Pushes the iFLAGS register as well as a return address to the stack. The return address is the second byte after the BRK instruction, meaning you can use the first byte after as a identifier or similar. Occupies opcode 59.

7.34 RETI

Returns from an interrupt. Pops its return address as well as the iFLAGS register off the stack to restore the previous state. Occupies opcode 5A.

Part IV

Designing

Chapter 8

Interrupts

A Cellia processor has 5 types of interrupts. An interrupt is a signal that, as its name implies, interrupts the normal flow of code and does something else. These interrupt types are as follows: Reset, Abort, Non-Maskable Interrupt, Interrupt Request, and Software Interrupt / Break. With the exception of the software interrupt or break, all of these are activated via active low hardware signals, meaning that the corresponding pin must be pulled towards ground in order to activate the interrupt. I will go over each of these in more details in this chapter.

8.1 Vectors

Every interrupt type requires a three byte vector indicating an absolute memory address to jump to when said interrupt occurs. Like all multi-byte values, these are stored in little endian format. However, unlike most multi-byte values, the direction flag does not apply to interrupt vectors, as the address read must be consistant. The vectors are as follows:

Byte	Page	Bank	Interrupt
FFFFF1	FFFFF2	FFFFF3	Break
FFFFF4	FFFFF5	FFFFF6	Interrupt Request
FFFFF7	FFFFF8	FFFFF9	Non-Maskable Interrupt
FFFFFA	FFFFFB	FFFFFC	Abort
FFFFFD	FFFFFE	FFFFFF	Reset

Each of these vectors point to an interrupt handler, also known as an Interrupt Service Routine (ISR). When any of these interrupts occur, the

processor reads these vectors, does what it needs to do, and jumps to the memory address the vectors point to and begins execution.

8.2 Reset

The Reset interrupt acts differently than most. The ISR the reset vector points to acts as the entry point to whatever firmware your system uses. Also, as its name suggests, it resets the processor completely when the Reset pin is grounded for a few clock cycles, then jumps to the ISR pointed to by the vector.

8.3 Abort

The Abort interrupt is intended to either halt or redirect program execution when a hardware exception occurs. It cannot be disabled.

8.4 Non-Maskable Interrupt

The Non-Maskable Interrupt, as it suggests, cannot be disabled by the Interrupt Disable flag. It is mainly intended for situations where the reason for the interrupt is so important that the processor cannot afford to wait before responding.

8.5 Interrupt Request

An Interrupt Request can have one of eight priority levels, from IRQ0 through IRQ7. The lower the number, the higher the priority. As such, when an interrupt request is received and the Interrupt Disable flag is cleared, the interrupt will go through and the lowest level interrupt is shown in the IL0 through IL3 bits of the iFLAGS register. IRQs are intended to be used by normal I/O devices to tell the processor "hey I have something I want you to look at".

8.6 Break

A Break or Software Interrupt occurs whenever the BRK instruction is executed. The main reason for this interrupt to exist is for things like system calls and stuff like that. As such, the BRK instruction pushes a return address that is two byte after the BRK itself, allowing you to put a single byte that will not be executed in the skipped address, which may be read by the Break ISR.

Chapter 9

Memory

The Cellia processor has a 24-bit memory address bus. This means that a Cellia system can have up to 16 mebibytes (16,384 kibibytes or 16,777,216 bytes) of memory. That includes both random access memory (aka working memory) and read only memory (aka instruction memory), as well as flash if you want non-volatile memory in your map.

Depending on what you're doing, you might only need ROM and no RAM or flash. However, if you plan to use the stack at all, you'll need at least 64 kibibytes (65,536 bytes) of RAM, or a bank of RAM, as the Stack can only work with RAM and it takes up an entire bank. Ideally, for a general purpose computer, you'll want much more than that.

Note that any Cellia processors I produce will not contain any circuitry for refreshing dynamic memory, as I assume most will be using static RAM. Most retro processors like the 6502 or 8088 didn't either, with the one exception I know of being the Z80, which integrated dynamic memory refresh circuitry. As such, you would either need to include your own refresh circuitry or (remember, I plan to have the processor core be open) get a processor card from someone who integrated DRAM refresh circuitry into their processor card. While this may seem to go against my first design goal, it actually doesn't. With the exception of the Z80 and modern processors, processors don't include DRAM refresh circuitry. While including DRAM refresh would make design easier, forgoing it doesn't make it any harder. Plus, it makes the actual design of the processor easier.

By the way, just gonna talk about the stack for a second here. There are generally four conventions for how the stack pointer works. Technically it's two and two, but combined they make four. There's empty stack vs full

stack, and ascending vs descending. Empty vs full refers to whhether the stack pointer points to the first address not in the stack (empty) versus the first address in the stack (full). These differ in how they work on pushes and pops. An empty stack moves after pushing and before popping, while a full stack does the opposite. As for the difference between ascending and descending, the pointer is increased on push and decreased on pop in an ascending stack, while its the exact opposite in a descending stack. While most architectures use an empty descending stack, Cellia will use an empty ascending stack. The stack pointer will start at hex 0000 of whatever bank you put it in and slowly progress to hex FFFF.

Chapter 10

I/O Devices

Cellia processors have a 16-bit port bus, allowing them to access 65,536 different I/O devices, assuming each device only uses one port. Granted that's never the case so it's less than that, but you get the point.

A Cellia processor cannot detect what kind of computer is in it, therefore it cannot determine what sorts of I/O devices said computer might have. Which is pretty standard for 8-bit and 16-bit processors. Although, a Cellia processor might also have integrated hardware, like the DRAM refresh discussed in the last chapter or things like audio and video processors.

I/O devices might need to interrupt the processor at some points. As the processor has eight interrupt levels, different I/O devices would be connected to each one. Lower levels have higher priority, so keep that in mind when connecting devices to interrupt levels. If you ever run out of interrupt levels, there are online designs for circuits that add extra levels.

Part V

The Future

Chapter 11

Implementing Cellia

Implementing Cellia is going to be a challenge, no matter how you look at it. While Verilog is a simple language to learn, like any sort of programming language (even if Verilog is technically a Hardware Design Language), taking the simple parts and using them to design something is always the hard part.

Firstly, what to implement it on? The most obvious answer is "FPGA", but then you have to ask what brand of FPGA? What manufacturer? How many logic elements do I need? Some of those questions are difficult to answer without diving in and testing things out. Others can be determined now.

To fit goal four, the FPGA I chose needed to work with open-source toolchains to fit the common open-source hardware definition. I did some research, and it seems the only open FPGA toolchain that I can get information about is SymbiFlow, which I first heard about when looking at the uCISC project. While it intends to eventually "compile" for all types of FPGAs, it currently only supports three: Xilinx 7-series (via Project X-Ray), Lattice iCE40 (via Project IceStorm (that sounds like an awesome video game codename)), and Lattice ECP5 (via Project Trellis).

As of now, SymbiFlow's Xilinx support is not as well as their Lattice support, as "Project X-Ray" still doesn't support DSPs and Hard Blocks, and is kinda meh with block RAM, while "Project IceStorm" and "Project Trellis" have checkmarks across the board on their website. So, time to go visit Lattice's website.

When looking at Lattice Semiconductor's lineup of FPGAs, ECP5 is listed under the "General Purpose" category, while iCE40 is listed under "Ultra Low Power". I ended up going with ECP5 as it has way more LUTs than

iCE40.

Now comes the question of which model of ECP5 to use. This is one of those questions where I can't really answer it yet. Each ECP5 has a different number of Look Up Tables, or LUTs, measured in the thousands, ranging from 12 to 84. I do have a general idea of how much I/O I would need, as I have designed a processor card connector pinout which I will discuss in the next chapter, which should help me choose the specific package the FPGA would be in.

I also plan to have four variants of Cellia processor. The main differences would be the size of the external bus (8-bit or 16-bit) and whether they would have integrated video and audio processors. So, 4 variants. The specs would be as follows:

		C100e	C100	C100AVe	C100AV
CPU	External Data Bus	8-bits	16-bits	8-bits	16-bits
	ALU Type	Integer	Integer	Integer	Integer
VPU	# Bits per RGB Channel	N/A	N/A	3	3
	# Bits Total	N/A	N/A	9	9
	# Colors Total	N/A	N/A	512	512
	# Colors on Screen	N/A	N/A	16	16
	Text Resolutions	N/A	N/A	20x15	20x15
				40x30	40x30
				80x60	80x60
	Graphics Resolutions	N/A	N/A	160x120	160x120
				320x240	320x240
				640x480	640x480
	# Sprites	N/A	N/A	32	32
	Sprite Resolutions	N/A	N/A	8x8	8x8
				16x16	16x16
	Color Depths	N/A	N/A	2-color	2-color
				4-color	4-color
				16-color	16-color
	VRAM Address Bus	N/A	N/A	18-bit	18-bit
APU	# Channels	N/A	N/A	Stereo	Stereo
	# PSG Voices per Channel	N/A	N/A	4	4
	# PSG Waveforms	N/A	N/A	Noise	Noise
				Sawtooth	Sawtooth
				Square	Square
				Triangle	Triangle
	ADSR Envelopes	N/A	N/A	Individual	Individual

Each of these variants could be on a different model of ECP5, depending on the needed LUTs and I/O. And, of course, once I implement the C100e and C100 variants I will open the architecture by uploading the verilog to the internet under an OHL. Oh, I also need to figure out what the maximum clock speed will be. But, that's for another day.

Chapter 12

Connector

All ECP5 FPGAs come in BGA packages of some sort. The main problem with that is that you can't really socket BGA chips, which goes against design goal three. Luckily, you can just solder the chip onto a daughter board that can be easily swapped out. When you take that into account, the main questions become "what form factor should we use" and "what's the connector and its pinout". I'll answer both of these questions in this chapter.

The easiest way to approach this would be to use a pre-existing form factor and connector. For example, a DDR2 SODIMM connector. This is the same connector and form factor used by Raspberry Pi Compute Modules, and it is what we will be using here. However, a Cellia processor card would not work in a board designed for an RPI Compute Module, as the pinouts are wholly different. While it's extremely future-proofed, here's the pinout... which is what I would say if LaTeXwould properly format a large table. Just take a look.

Front Pin	Name	Description	Back Pin	Name	Description
1	GND	Ground	2	GND	Ground
3	GND	Ground	4	GND	Ground
5	D0	Data Bit 0	6	VD0	VRAM Da
7	D1	Data Bit 1	8	VD1	VRAM Da
9	D2	Data Bit 2	10	VD2	VRAM Da
11	D3	Data BIt 3	12	VD3	VRAM Da
13	D4	Data Bit 4	14	VD4	VRAM Da
15	D5	Data Bit 5	16	VD5	VRAM Da
17	D6	Data Bit 6	18	VD6	VRAM Da
19	D7	Data Bit 7	20	VD7	VRAM Da
21	D8	Data Bit 8	22	VD8	VRAM Da
23	D9	Data Bit 9	24	VD9	VRAM Da
25	D10	Data Bit 10	26	VD10	VRAM Da
27	D11	Data Bit 11	28	VD11	VRAM Da
29	D12	Data Bit 12	30	VD12	VRAM Da
31	D13	Data Bit 13	32	VD13	VRAM Da
33	D14	Data Bit 14	34	VD15	VRAM Da
35	D15	Data Bit 15	36	VD16	VRAM Da
37	GND	Ground	38	GND	Ground
39	GND	Ground	40	GND	Ground
KEY NOTCH					
41	GND	Ground	42	GND	Ground
43	PWR	Power	44	GND	Ground
45	GND	Ground	46	VA0	VRAM Ad
47	GND	Ground	48	VA1	VRAM Ad
49	A0	Address Bit 0	50	VA2	VRAM Ad
51	A1	Address Bit 1	52	VA3	VRAM Ad
53	A2	Address Bit 2	54	VA4	VRAM Ad
55	A3	Address Bit 3	56	VA5	VRAM Ad
57	A4	Address Bit 4	58	VA6	VRAM Ad
59	A5	Address Bit 5	60	VA7	VRAM Ad
61	A6	Address Bit 6	62	VA8	VRAM Ad
63	A7	Address Bit 7	64	VA9	VRAM Ad
65	A8	Address Bit 8	66	VA10	VRAM Ad
67	A9	Address Bit 9	68	VA11	VRAM Ad
69	A10	Address Bit 10	70	VA12	VRAM Ad
71	A11	Address Bit 11	72	VA13	VRAM Ad
73	A12	Address Bit 12	74	VA14	VRAM Ad
75	A13	Address Bit 13	76	VA15	VRAM Ad
77	A14	Address Bit 14	78	VA16	VRAM Ad
79	A15	Address Bit 15	80	VA17	VRAM Ad
81	A16	Address Bit 16	82	VA18	VRAM Ad
83	A17	Address Bit 17	84	VA19	VRAM Ad
85	A18	Address Bit 18	86	VA20	VRAM Ad
87	A19	Address Bit 19	88	VA21	VRAM Ad
89	A20	Address Bit 20	90	VA22	VRAM Ad
91	A21	Address Bit 21	92	VA23	VRAM Ad
93	A22	Address Bit 22	94	GND	Ground

Yeah. If you want to see a better version (aka have me send you a copy of the spreadsheet or the .tex file this PDF or book originated from), contact me. I'll probably have my email address listed somewhere but if I don't, there's a contact form on my website located at ashtons.xyz.

Chapter 13

Future Expansion

Obviously there's some space for expanding the architecture. Everything from adding 16-bit floating point math to adding multi-core support. I'll briefly go over how these features might be implemented.

13.1 Floating Point

Pretty much every processor that can do floating point math has special floating point registers. As I purposefully had only 8-ish general integer registers, I can have another 8 general floating point registers. These would be labelled as fA, fB, fC, fD, fW, fX, fY, and fZ. The reason I didn't have 16 general integer registers was to avoid having to add an extra move instruction just for floating point registers.

There would also have to be a floating-point specific flags register, which would be called fFLAGS (to go along with iFLAGS). It would also be 16-bits long, even if most of that would be unused.

By the way, 16-bit floating point is also called half-precision, as 32-bit is single-presicion and 16-bit is half of that. The standard format for 16-bit floating point is as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F

S stands for "sign", E stands for "exponent", and F stands for "fraction". Yes I got this from Wikipedia. This is the standard IEEE 754 half-precision / binary16 floating point format. The exponent uses offset-binary, with the offset or exponent bias being 15. This means that 15 must be subtracted

from the exponent to get the true exponent.

By the way, there is a way to encode infinity and NaN (not a number). Setting the exponent to binary 11111, having the "fraction" be zero makes a representation for positive or negative infinity. If the "fraction" isn't zero, then that represents NaN.

Beyond the registers, I would need to implement new arithmetic, comparison, and branch instructions, like an ADDF or SUBF, as well as CMPF, TSFF, BEQF, BNEF, BLTF, BGTF, BLEF, BGEF.

13.2 Multiple Cores

I'm going to be real here, I have no idea how I would implement this. Threads seem to be easier - you have one core with one ALU, but two execution pipelines. Adding multiple cores however? That's a bit more complicated. How do the cores communicate? Which core handles the boot process? How do you divy up work between the cores? So many questions and not enough answers.

Last Note

Eventually, I plan to create an open-source Cellia cross-assembler, called chasm. Why chasm? Because Cellia starts with a C, and "casm" just didn't look right to me. So, chasm. By default it will spit out raw binary, suitable for flashing onto a ROM of some sort. However, if anyone makes an operating system that runs on Cellia that has a special program format, I plan to implement an API(?) that would allow you to define your special format and have it work with chasm. At the moment I don't know what language I'll write chasm in. A common thing I have heard is that if its interpreted you need to write it in a compiled language and if its compiled/assembled you can write it in an interpreted language. So, maybe Python? Or Rust? Ehhh, I still need to work on my Rust skills.

While I should put contact info here, I don't exactly want to get spammed so just go to my website which was previously stated and use the contact form on there, as it has a captcha. Also I have a youtube so :)

(By the way - turns out I might be only person who is named Ashton Snapp, at least in the US)