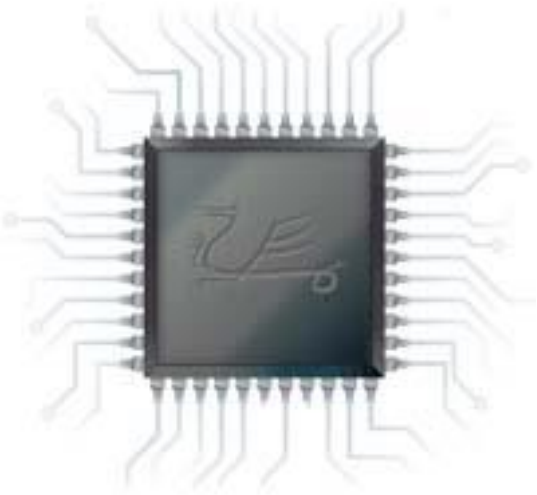
Processor Design



Simon Southwell

October 2022

Preface

This document brings together four articles written in September and October 2022 and published on LinkedIn, that is a look at processor architecture and logic design, including programming with assembly language. It uses the RISC-V architecture as a case study and for the Verilog implementation and ISS that accompany the articles.

Simon Southwell
Cambridge, UK
September 2022

Contents

Part 1: Processor Overview

Introduction

This and the next few articles are based on the notes I made for a mentoring program where I covered processor architecture and logic design using RISC-V as the case-study, as this is a modern RISC based instruction set architecture, is open-source, and is making a lot of noise in the industry right now. From knowing nothing about how processor worked the mentee executed a fully working implementation that passed all the relevant RISC-V International instruction tests. They then went on, of their own volition, to pipeline it for single cycle operation on non-memory instructions. I'm hoping that this article will provide enough information that anyone who wants to do so can reproduce what my mentee did, at least to a finite state-machine based design, but I will also discuss some steps beyond these fundamental principles for more advanced features.

RISC-V is being used as a relevant example instruction set, but this is not a document on all aspects of RISC-V and I will stick to only those features that allow a processor core to be implemented. Thus, in this article, we will stick to the base implementation and relevant control and status registers, whereas there are many instruction extensions to this base. Also, RISC-V can be 32- or 64-bit (or even 128-bit) and has three privilege modes—machine, supervisor, and user—or four if you count hypervisor—but we will stick to 32-bits and the highest priority mode only (machine) as that has restrictions on any permissions. RISC-V also supports multiple hardware threads (*harts*), but we will stick to just one…and so on. There are many, many good resources out there for those who want to know more about RISC-V, including the specifications which I will give links to at the end of the article.

Throughout the articles I will be making reference to my own RISC-V logic implementation for illustration and example. The source code, along with documentation, is available on github. It is targeted at FPGA, though would easily be implemented on an ASIC, and is restricted to the specification I have just laid down. I have sometimes sacrificed efficiency for clarity in the design as it is meant for informative and educative purposes and there are many better, more developed, more verified, implementations than this available as open source (e.g., Ibex), but my core is architected and documented for ease of understanding. Where a 'better' approach might be warranted I will discuss this in the text to explore more general processor design features.

In this first article, though, is discussed processor function and design in a generic way (though looking at some real examples) in order to define what a processor is, where it fits within a system, and what the common traits are for the vast majority of processors. In future articles, we will look at the RISC-V architecture, the instructions it defines for the base system, and the register sets (both general purpose and CSR). Then we will look at the logic architecture to actually implement such a processor core, including optimisations and alternatives. Finally, we will look at assembly language, the lowest level programming (discounting programming directly in machine code, which nobody would be foolish to do since the 1970s). There is an instruction set simulator (ISS) as part of the accompanying RISC-V project, and this can be used to experiment with assembly language programming without the need for processor hardware.

What is a Processor?

Firstly, I want to say that a processor doesn't do very much. It reads a set of fairly simple instructions from a memory or internal registers, manipulates associated data according to those instructions, and stores this altered data either internally, or back to memory. And that's it. The power comes from the fact that it can do these instructions very fast, and that more complex operations can be achieved by combining the limited set of instructions. The instructions used vary between different processors, but a result that might surprise you is that, in the limit, only one instruction is really necessary! All the processors that have multiple instructions are actually doing engineering to make the processors' operations more efficient. Such One Instruction Set Computers (OISC) actually exist and can perform the same functions as a bigger processor, albeit much less efficiently.

A modern processor has more than one instruction, and these are encoded as plain binary numbers ('machine code') which the processor reads from memory (maybe RAM, flash, or other storage device) to manipulate data and read and write to memory. This is the software running on the processor. It will have a bus to access memory and other devices such as I/O, using protocols such as AXI or Avalon (see my article on busses). The internal registers vary in number and purpose between different processors, as we shall see, and this register set and the instruction set that the processor recognises, is known as the processor's Instruction Set Architecture (ISA). The ISA defines, then, whether the processor is RISC-V RV32I, ARMv8, IA64 etc. There may be different implementations of the same ISA, but the processor is classed based on the ISA that it implements.

Processors are often classed a *n* bit, for example 8-bit. This (usually) refers to the size of the data and instructions it processes, and this has been increasing with time. For

microprocessors, it all started with the Intel 4004 at 4-bit, and the 1980s 8-bit home computer revolution with 8-bit processors, such as the MOS 6502, Zilog Z80, and the Intel 8080—all originally designed in the 1970s. A short period of 16-bit processors (e.g., Motorola 68000 and Intel 80286) was taken over by 32-bit processors such as the Intel 80386, SPARC v8 and ARM Cortex-M. This 32-bit era still persists in the embedded processor world, but modern PCs, workstations, and smart phones, use 64-bit processors (I'm ignoring graphics processors), such as Intel i9, and the Apple A15 incorporating 64-bit ARM based processors. Beyond this are Very Long Instruction Word (VLIW) processors such as the HP/STMicroelectronics ST200 family of processors, Analog Device's SHARD DSP processor and the u-blox software defined model (SDM) processor (which I worked on as a DSP software engineer doing 4G physical layer code including the code on the VLIW processor).

CISC versus RISC

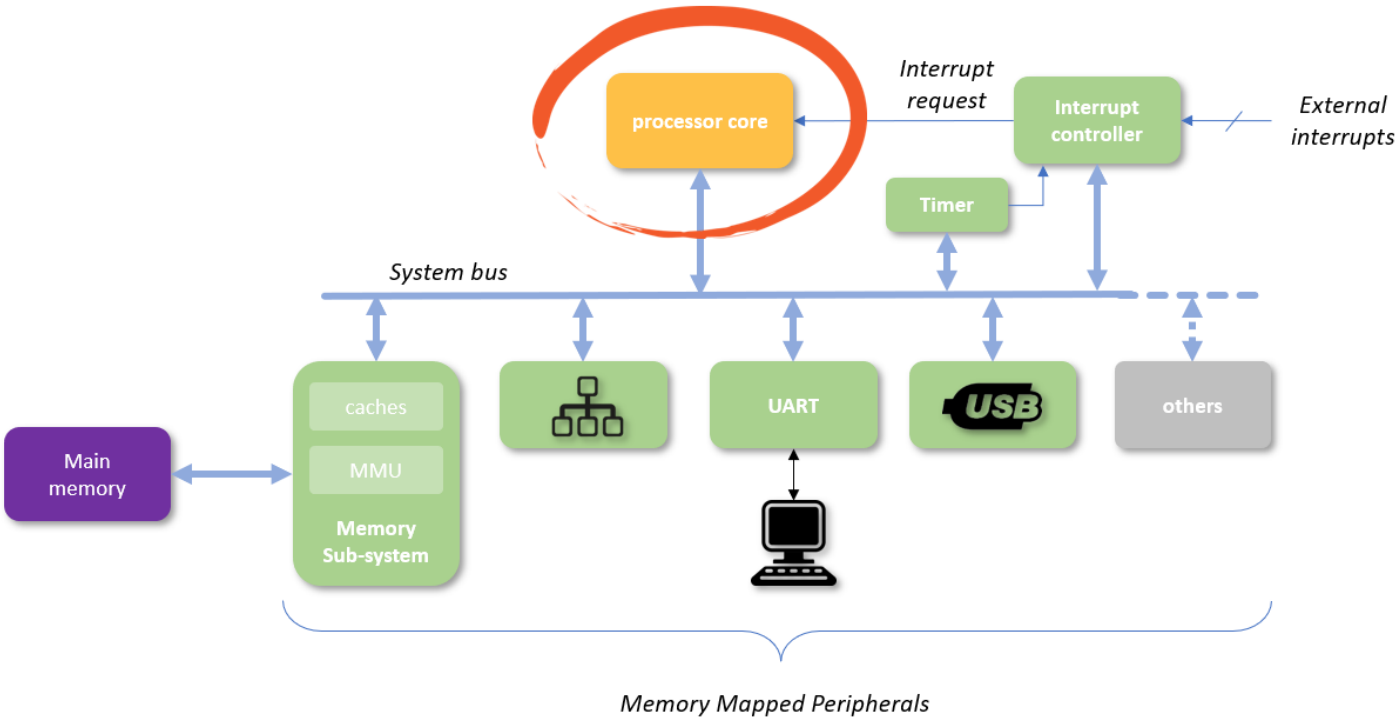
Another categorisation of processors is whether it is CISC or RISC. That is, is it a complex instruction set computer, or a reduced instruction set computer? In the early days of processors, iterations of processors tended to add more instructions with more complex functionality to aid in the efficiency of programs which, originally, were written directly with the processors' own specific instructions. With the advent of higher-level programming languages and their compilers, it was found in research that, in general, compilers would use 80% of the instructions only 20% of the time, and 20% of the instructions 80% of the time. Using this result, work was done to design processor architectures that had fewer (i.e., a reduced) number of instructions—the ones that ran most of the time—that run more efficiently. The more complex functions can be emulated using multiple simpler instructions which, although slower than a dedicated instruction, is done less often on a processor that can run much faster.

The term complex instruction set computer (CISC) was retro-fitted to earlier processors that had characteristics of large instructions set, including instructions that had complex functionality, with multiple 'modes' for each instruction and could be variable in the size to encode the instruction. An example of a CISC processor is the Intel x86 family and the processors derived from this architecture. RISC processors, by contrast, have fewer more simple instructions, implemented for fast execution. The instructions are all fixed width allowing ease of pipelining an implementation. Also, RISC processors generally separate data manipulation from memory input and output. So all data manipulation is done from values held in internal registers and placed back in internal registers. All movement to or from memory are done with instructions that can't alter the data. This separation avoids

having multiple modes for each data manipulation instruction—e.g., supporting a function that can have data either from memory or a register, or a memory location indirected by another register etc., as is common in CISC processors. Example RISC processor architectures include ARM and, of course, RISC-V. Indeed, even modern CISC processors often have an internal RISC architecture, and stages to break down complex instructions to multiple simpler internal operations.

Role of a Processor in an Embedded System

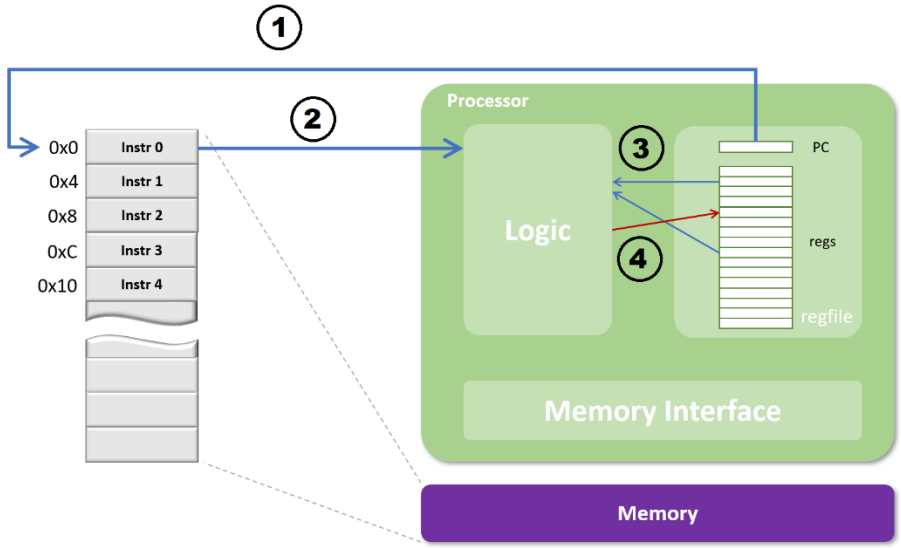
My particular interests and experiences are with embedded systems and SoCs, so where does a processor fit in such a system? Within an embedded system or SoC one or more processors might sit on a system bus, such as an AHB or AXI bus, along with a memory sub-system (with caches and MMU) and with peripherals that it might control, such as ethernet, UART, USB etc. It might also have an interrupt controller and timer to produce internal and external 'events' (more on this later). The diagram below shows a simple SoC arrangement.



This arrangement is oversimplified, but indicative of most processor-based systems, with software in memory running on one or more processors which control a set of peripheral devices over the system bus or interconnect fabric, which make up a system, such a controller for a storage device, a smartphone, or even a Raspberry Pi. If there are multiple cores, then the cores may also communicate with each other, usually through memory.

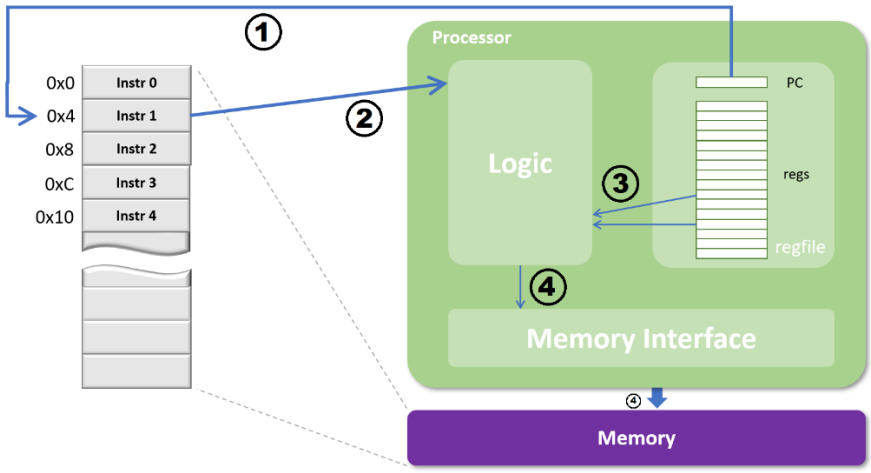
Basic CPU Operation

We've not yet defined any kinds of instructions that a processor may use, but we can still discuss what a processor does when it is powered up and taken out of reset that is common to the majority of processors. The diagram below shows what happens at this first step.



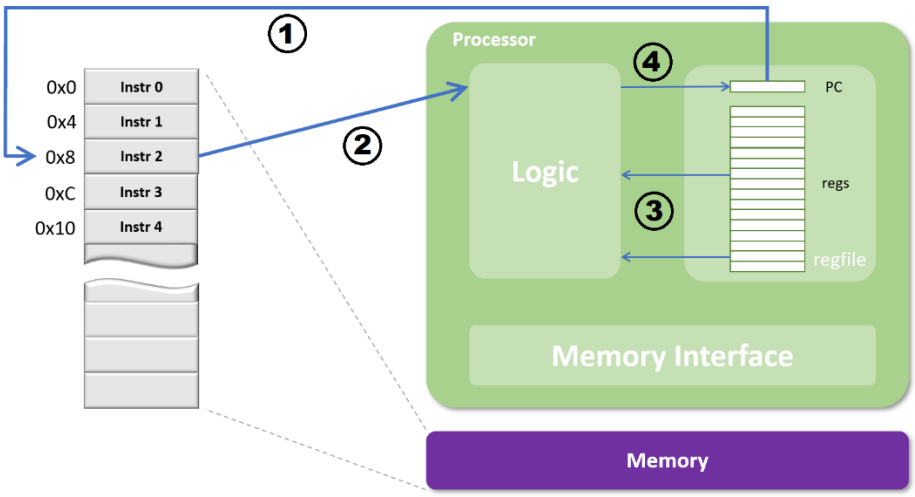
After reset is removed, a processor will start to read an instruction from some predetermined fixed location, for example at address 0 (shown as step 1 on the diagram). The value of the instruction is returned from memory to the logic for interpretation (step 2). It may be that the particular instruction manipulates data from two of the internal registers and so these are fetched by the logic (step 3). The result of that manipulation might be, depending on the instruction, placed back into another internal register (step 4), and the instruction execution is completed. In step 1, the address is shown to come from a particular register labelled PC. This is the program counter. When an instruction completes, this is normally incremented to point to the start address of the next instruction located in memory immediately after that just executed. For a 32-bit RISC machine, all instructions are 32-bits, or 4 bytes, and so the PC (a byte address) would be incremented by 4 and the whole cycle started again.

The next instruction, instead of manipulating data, might be a memory access, such as a write.



The diagram shows a store operation, where two internal register values are accessed, one to form an address for the data to be written and one for the actual data to be stored. So, instead of the 'result' being written back to an internal register, it is directed to memory. Similarly, a read from memory instruction might read an internal register for an address value, perform a read operation from memory, and the returned data written back to another internal register.

One last class of instruction is one which can change the value of the PC from its default of moving to the address of the next instruction. The diagram below illustrates this:



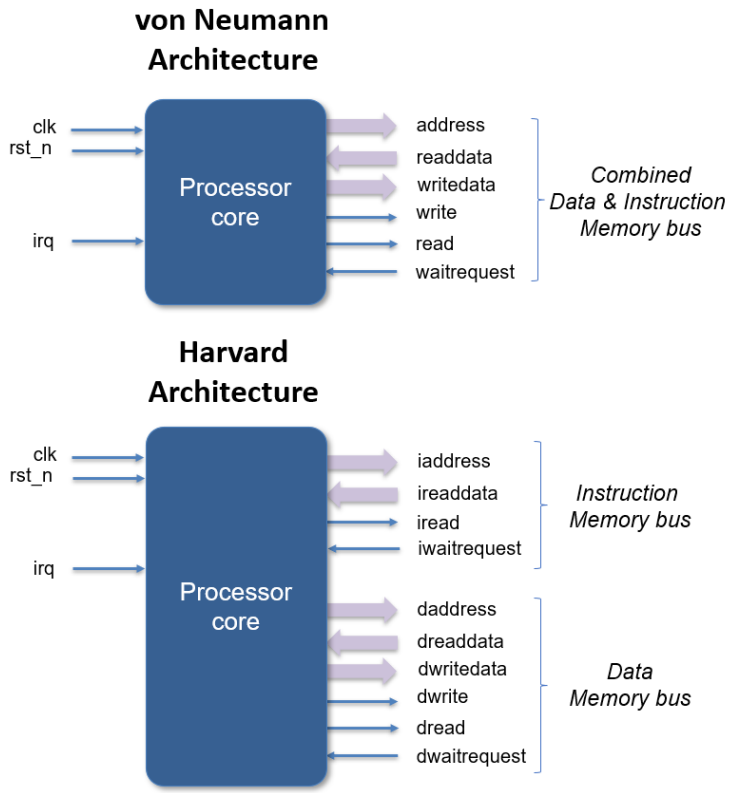
Here, the instruction is read from the current PC address and sent to the logic. The logic might read two internal registers to compare their values. If they meet some criteria, such as being equal, the logic then overrides the default increment of the PC to some new address location. Often this is an offset from the current PC value, forward or back, that is encoded in the instruction itself. So, in the diagram, if the two registers did not meet the criteria (of being equal, say), then the PC would increment as normal and the new PC address would be `0xC`. If the two register were equal, then the instruction might have an offset of, say, -8 and the PC would be set back to `0x0`

and start executing instruction from there once more. The offset could just as easily have been positive, +8, say, and would then skip the instructions at `0xC` and `0x10`, and start executing instructions from `0x14`.

This basically describes all the types of operations that a modern RISC processor does: data manipulation to and from registers, memory reads and writes, and overriding the program counter. The only addition to this is an exception where an external signal (interrupt) or an internal error event can also change the program counter value, but this is not normal program flow and we will look at this shortly.

Processor Core Port Architecture

Above has been shown that a processor core reads instructions from memory. Some of those instructions will direct the core to load data from memory or store data to memory. These two classes of memory access, instruction and data, can be handled in one of two ways via the external ports of the core. The diagram below shows two examples of cores with memory port configurations.



The first configuration has a single memory port. The diagram shows a simple SRAM like port with a wait request, but it could be any kind of port to access memory (e.g., AHB). This configuration is known as a von Neumann architecture. This might cause conflict on memory access if an instruction could be read whilst data is being loaded from, or stored to, memory. If the internal design is such that an instruction is

completed before the next instruction is fetched, then no conflict arises. However, this is not a very efficient implementation, and modern pipelined implementations can fetch instructions in parallel with accessing memory for data. The second configuration has separate memory ports for instructions (a read only port) and for data (a read and write port) and is known as a Harvard architecture. Now, memory can be accessed for data as well as instructions fetched in parallel. It may be that the instructions are in a separate ROM which is connected directly to the instruction port, with the data port to RAM or DRAM. Alternatively, the instructions and data may ultimately reside in the same set of memory, which would simply move the conflict to the memory sub-system which would need to arbitrate access for both ports. However, the memory bandwidth may be higher than that of the core's memory ports, and so both ports can be run at 100% efficiency. This is not an uncommon situation in an SoC.

Internal Registers

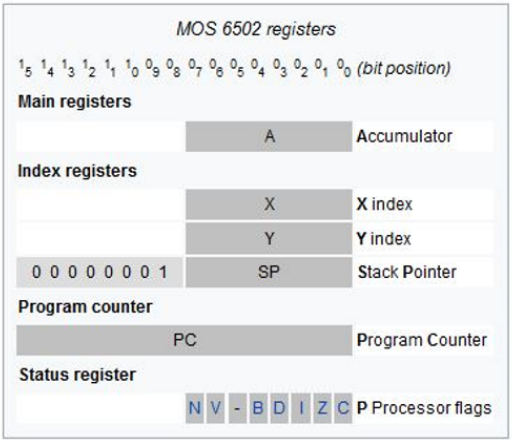
Also common amongst nearly all processors is a set of internal registers, which we have alluded to already. The type and number vary, but they all have very similar roles. The basic categories of these internal registers are as follows:

- Program counter (PC)
- Control and status registers (CSR)
- General purpose registers

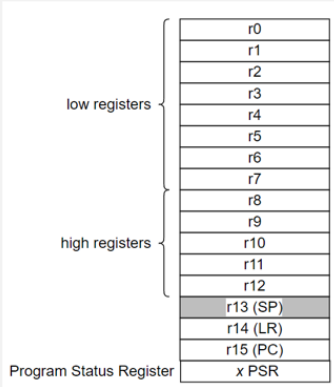
Within the general-purpose registers, certain ones might be nominated as having additional specific purposes but could still be used as a general-purpose register. Also, many modern processors nominate a register to always read as zero.

Below are some examples or real registers sets from three different processors, and from different eras.

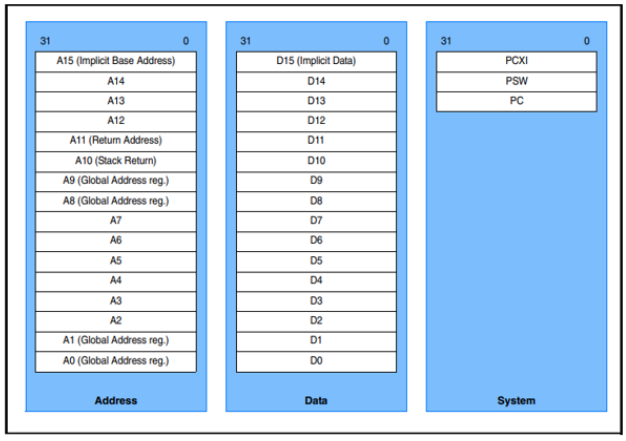
MOS 6502 registers



ARM registers



Infineon Tricore registers



All three of these examples have registers that fit, roughly, into the listed categories. The 6502 has A, X and Y registers that are more or less general purpose. The ‘stack pointer’ is kind of custom but could be used as a general-purpose register. The PC maps directly to the functionality already described, and the P register (processor flags) is a status register. The ARM processor has a set of general-purpose registers, r0 to r15, where r13 is nominated for a ‘stack pointer’, r14 for a ‘link register’ and r15 for the program counter. The PSR is the status register. Finally, the Tricore processor from Infineon (which I worked on, constructing software models of the processor and system) has 32 general purpose register (though split as address and data) and a PC, with control and status in the PCXI and PSW registers.

This, I think, serves to illustrate that there is a commonality of internal registers amongst varied processors. Compilers, such as gcc, work (in broad terms) by having a generic model of registers which it uses to map the programming language code to an intermediate code before mapping to actual registers (and instructions) available on the target processor. Modern processor architecture is now sympathetic to this compilation process to make mapping straightforward.

In the diagram for the three processor register sets, looking at the ARM and Tricore, the diagrams show labels in brackets next to some of the register names. I've mentioned before that some registers are nominated for particular functions but this need not be the case (though it might be for some processors). In reality, only if writing code in assembly language (the lowest level programming) would this be true. The reason for nominating registers like this is to have a convention when compiling code from a higher-level language. For interoperability of code compiled separately it is helpful if everyone follows the same convention. This convention is often called the application binary interface (ABI). It dictates things like the register to use for the stack pointer (I'm not going to define these terms here), which register to use as inputs to the call of a function, and which as outputs and things like this, relevant to a high-level language.

Exceptions and Interrupts

Before we leave this generic discussion on how processors are used and operate, I want to mention exceptions and interrupts. I have mentioned these in passing in the above sections, but I want to fill in some of the blanks.

We have discussed the flow of a program running on a processor as proceeding in a sequential fashion through memory, though this can be altered using instructions to change the default program counter increment to change the flow through the running program. Another way to change the flow of the program is with 'exceptions'. These usually happen from some internal error condition, or from some external event or interrupt. In some cases an instruction can intentionally cause an exception.

For the internal events an error condition might be, for example, an unrecognised instruction. If an instruction read from memory has a value that does not decode to any known instruction, or some fields within an instruction have illegal values, this is an error and causes an exception. Some processors have special instructions to actually cause particular exceptions, so that a program can generate these events itself, rather than on an error condition.

For external events, these usually come in the form of interrupt input signals. There can be multiple interrupt inputs to a processor core, but fundamentally this boils down to one exception with other logic (an interrupt controller) sorting out if a given interrupt input is enabled (so the processor responds) or is the top priority to respond to if multiple interrupts are active. The interrupt controller logic might be within a core, but is often external to the core, with just a single input signal to the processor.

For both the internal and external events a processor will finish its current instruction and then change the program counter from its normal next value to be some fixed, predetermined, address in memory (not unlike after coming out of reset). It will save the address of the next instruction it would have normally executed to some store (usually a register). In some cases, the source of the exception may add an offset from the fixed address to differentiate the various types and sources of exceptions. The code that is located at this fixed region of memory is specially written and is known as an exception handler. So, for example, if a UART peripheral has a new byte just arrived it might raise an interrupt so indicate this byte needs processing. The interrupt causes an exception, the exception handler is called and will identify that the exception is from the UART, so call a routine to fetch the byte which might place it in a buffer in memory for the main software to process. When this is finished (and the handler code is usually as small and fast as possible), the exception can be completed and the program counter reset to the saved address so that it can carry on from where it left off at the point of the exception. If the event handler can't process the exception, this is when it can 'crash', perhaps displaying a message (if it can) before halting the processor.

In the simplified SoC diagram above I shown a timer, along with the other peripherals which, itself, can be a source of interrupts. This is important when constructing a multi-tasking system, where the processor core is running an operating system and several other processes and threads 'concurrently'. That is, it appears that multiple programs are running, but actually only one is running at a time, and they are swapped out at regular intervals by the operating system software, which is where the timer comes in. This might be programmed to interrupt after a given time, and then a particular process's code allowed to run. When the timer interrupts at the end of the period, the exception handler notices that this is so, and hands control back to the operating system, which can then swap in another process to start running from where it last was running when it was swapped out, and so on. (There are other reasons a process might be swapped out, such as it waiting for the UART to receive another byte, as it won't make progress anyway, so the OS might as well let something else run, but this is still event driven and under the control of the OS software.) Thus multiple processes and threads can make progress on a single core, as if running in parallel on multiple processors. However, from the processor core's point of view, this is all just interrupts to jump to the handlers, and the handlers are software to be run, just as for the main code, to know what actions need taking.

Conclusions

In this introductory article we have restricted to discussing processor cores in a generic way, without committing to details of supported instructions (which can be just one, but usually isn't) or the particulars of a logic implementation. This is so that the fundamental characteristics can be teased out, without the complexity of the details for a particular architecture or an implementation strategy.

And it turns out that a processor does not do very much that's particularly complicated (I hope I've demonstrated). It reads instructions (which are just numbers in memory) which tells the logic to process data in and out of internal registers, load or store data from memory or update the program counter to start executing somewhere other than the next sequential location. Exceptions are very much like the PC update instructions, except they are caused by errors or external events (interrupts) or maybe even special instructions. In these cases, the new program counter address is set to a fixed location (or fixed location plus an offset), and normal program flow can be restored once the exception is handled by the specially written software, having saved the place where the exception changed the PC.

In the next article I want to map what we have discussed here to a modern real-world example, the RISC-V architecture. We will look at the base configuration, the instructions defined for this, and the registers, both general purpose and the control and status registers. This should set us up nicely for discussing an implementation in logic.

Part 2: Introduction to RISC-V

Introduction

In the first part of this document I discussed what a processor was and what it did in generic terms, with just some loose examples of particular processors. In order to get to implementing a processor in logic, more specific details are required and I said we'd look at the base specification for RISC-V as a relevant modern processor instruction set architecture, whose ISA is open-source. In this article, then, I will introduce the instruction set for the RV32I specification—that is, the 32-bit integer specification that's the minimum feature set for compliance. Also, to tie in with the other generic processor aspects discussed in Part 1, the control and status registers (CSRs) will be introduced, though, strictly speaking, this is an extension (*Zicsr*) to the base specification, but all practical implementations would have these registers. You'll be glad to hear that there is only a small minimum set that need to be implemented. Following on from the CSR registers, but closely associated with them, is the RISC-V exception handling—events and interrupts—which will be discussed.

In order to strip away obfuscating and unnecessary detail, but still have an operational and compliant processor, other aspects of RISC-V will default to the simplest legal configuration. All the extensions (except *Zicsr*) will not be detailed, though we will have a review of what the common extensions are and what they do, and we will stick to the highest-level privilege mode (machine). RISC-V also defines *harts*, which are hardware threads—multiple copies of the processor registers for fast, hardware assisted, swapping between different program flows. The minimum number specified is 1, so we will stick this. The hope, here is that, by sticking to a minimal but functional and compliant implementation (when we get to the logic), this becomes a foundation for discovering the additional features that can augment the base specification without too much trouble.

I realise that I am writing this article as summary of a specification and so, necessarily, this might be a little repetitive in form, but I hope to map back to the concepts introduced in the first part of the document as we go, and I promise we are not going through every instruction in minute detail (not that there are too many). The article is a little longer than my normal articles, but we need to cover enough ground to have as a basis for an implementation.

RISC-V Register set

Before looking at the RV32I instruction set, the processor internal general-purpose register set needs to be defined. The tables below list these registers:

register	ABI	description
x0	zero	Hardwired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary 0
x6	t1	Temporary 1
x7	t2	Temporary 2
x8	s0/fp	Saved register0/frame pointer
x9	s1	Saved register 1
x10	a0	Function argument/return value 0
x11	a1	Function argument/return value 1
x12	a2	Function argument 2
x13	a3	Function argument 3
X14	a4	Function argument 4
x15	a5	Function argument 5

register	ABI	description
x16	a6	Function argument 6
x17	a7	Function argument 7
x18	s2	Saved register 2
x19	s3	Saved register 3
x20	s4	Saved register 4
x21	s5	Saved register 5
x22	s6	Saved register 6
x23	s7	Saved register 7
x24	s8	Saved register 8
x25	s9	Saved register 9
x26	s10	Saved register 10
x27	s11	Saved register 11
x28	t3	Temporary 3
x29	t4	Temporary 4
x30	t5	Temporary 5
x31	t6	Temporary 6
pc	-	Program counter

So, RV32I defines 32 general-purpose registers from *x0* to *x31*, plus a program counter (PC). There is nothing special about the general-purpose registers except for *x0*. This register will always return 0 when read. It can be written to, but its value will remain at zero. For *x1* to *x31*, the specification does not define the values of these registers after reset, and these may be written and read using the instructions we will define presently.

The tables, next to the register column, define a name for each register under ABI. In Part 1, this was mentioned as the application binary interface, which is a convention on the use of registers for programming languages. As this is academic to a logic implementation, the ABI names are just here for reference and because, when we get to assembly language programming either the register name or the ABI name can be used, and disassemblers can also be configured to use one or the other.

All the registers are 32-bits wide since we are looking at the 32-bit specification. If this were for the 64-bit specification (RV64I) than these would all be 64 bits but would still have the same names. It is this register set, then, on which the processor's instructions will perform operations.

RV32I Instructions

As mentioned before, RISC processors have the characteristic of a limited number of fixed width instructions. For RV32I these are all 32-bit instructions. Let's remind ourselves of the categories of processor instructions defined in general in Part 1.

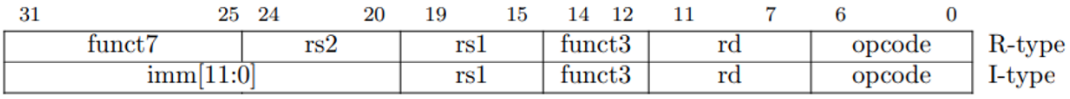
- Instructions for data manipulation in and out of internal registers
- Instructions for reading and writing memory
- Instructions for altering the default update of the PC
- Instructions to generate events and other system related functions

The RISC-V RV32I specification follows this model, and in the rest of this section we will look at the instructions for each of them.

Instruction Formats

Before looking at specific instructions, I mentioned in Part 1 that the instructions are just binary numbers (32-bits for RV32I), and these are known as machine code. Machine code instructions are sub-divided into bit fields with different meanings associated with them. Some of the fields are used to identify the unique instruction (thus its operation), whilst others are for identifying which registers to read from and to write back a result. Some fields are even just encoded constant numbers to be used in an operation.

For the majority of the data manipulation instructions these instructions follow one of two formats, the R-type or I-type format, as shown below:



Both these instruction format types share common attributes. The 7-bit '*opcode*' (operation code) is the first field that identifies a 'class' of instruction. For example, all arithmetic instructions of R-type format have the same opcode value (in this case `0110011`b). The next field is *rd*, which is a five-bit index into the register set, and is the destination register—i.e., the register (*x0* to *x31*) that the result of the operation will be written. The 3-bit function field (*funct3*) further refines the specific instruction within the class (e.g., add, subtract etc.) and, for R-type format instructions, it also requires the 7-bit *funct7* field, whereas the I-type format instructions are uniquely identified with just *funct3*. The *rs1* field is a 5-bit index into the processor registers from where a value will be read for manipulation. For R-type instructions there is also an *rs2* index for reading a second value from a register to process. For I-Type instruction, there is no second register index, but a field called '*imm*'. This 12-bit field

stands for ‘immediate’ and is a number encoded within the instruction itself in place of a number fetched from an internal register. This number can be interpreted as a signed value or an unsigned value, depending on the specific instruction, but is limited in range; to either -2048 to 2047, or 0 to 4095.

There are only four other instructions formats, and they have the exact same sorts of fields as the R- and I-type formats, but just formatted differently or, in the case of immediate fields, a different number and range of bits. These remaining formats are shown below.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[20|10:1|11|19:12]				rd	opcode	J-type
imm[31:12]				rd	opcode	U-type

The S-Type format is used exclusively for memory store instructions (load instructions use the I-type format). The main difference here is that the immediate value is split up into different bits. The RISC-V specification will always try to keep formats that have a particular field, such as *rd*, in the same bits for all of the different types. This makes decoding so much easier (and faster). Something has to give, though, and the immediate values tend to get jumbled up.

The B-type and J-type formats are for the PC updating instructions (branch and jump). The immediate values look jumbled up somewhat but, even here, an attempt was made to align the same bit position in the same locations within the instruction. So, for example, the B-Type instruction does not require *imm*[0], so *imm*[11] is placed in this space, leaving *imm*[4:1] in the same positions as for the S-type format. Similarly, for *imm*[10:5]. This bit alignment is followed where possible for J-type instructions as well.

The final format is the U-type, used for a couple of special data manipulation instructions to load a 20-bit immediate value into the upper 20 bits of a register, used for forming addresses within a register. Thus, only a destination register index (*rd*) and the 20 bit immediate number are required after the *opcode*. No attempt is made to align these immediate bits with the other formats.

Arithmetic Instructions

Now (finally!) we can look at some instructions. This first set is for arithmetic operations. In this context this not only means basic arithmetic (adding and subtracting) but also shifting, comparison and bitwise operations. The table below shows the arithmetic operations for RV32I that have the R-type format.

name	31:25	24:20	19:15	14:12	11:7	opcode	Description
add	0000000	rs2	rs1	000	rd	0110011	(rd) = (rs1) + (rs2)
sub	0100000	rs2	rs1	000	rd	0110011	(rd) = (rs1) - (rs2)
slt	0000000	rs2	rs1	010	rd	0110011	(rd) = signed(rs1) < signed(rs2) ? 1 : 0
sltu	0000000	rs2	rs1	011	rd	0110011	(rd) = unsigned(rs1) < unsigned(rs2) ? 1 : 0
xor	0000000	rs2	rs1	100	rd	0110011	(rd) = (rs1) ^ (rs2)
or	0000000	rs2	rs1	110	rd	0110011	(rd) = (rs1) | (rs2)
and	0000000	rs2	rs1	111	rd	0110011	(rd) = (rs1) & (rs2)
sll	0000000	rs2	rs1	001	rd	0110011	(rd) = (rs1) << unsigned(rs2[4:0])
srl	0000000	rs2	rs1	101	rd	0110011	(rd) = (rs1) >> unsigned(rs2[4:0])
sra	0100000	rs2	rs1	101	rd	0110011	(rd) = signed(rs1) >>> unsigned(rs2[4:0])

Since these are R-type instructions, there are two source register indexes (*rs1* and *rs2*) and a destination register index (*rd*). The values for the *opcode*, *funct3* and *funct7* are shown for each unique instruction. The description shows the operation for each instruction, where a register index in brackets means the actual register, such as *x17*. It should be noted at this point that the destination register index can be the same as one of the source registers' and, indeed, the two source register indexes could also be the same. This is perfectly legal.

The first two operations are add (**add**) and subtract (**sub**) and the values of the two source registers are added or subtracted and placed in the destination register. For these two operations, whether the two source values are interpreted as signed or unsigned does not affect the result and is the same operation.

The next two instruction are comparison instructions: 'set if less than'. The signs of the values do matter here, so there are signed (**slt**) and unsigned (**sltu**) versions of the instruction. If the *rs1* register is less than the *rs2* register, then the *rd* register is set to 1, else it is set to 0.

The next three instructions are for logic bit manipulation, with '**and'**, '**or**' and '**xor**' operations. Since these instructions just do logic operations across the 32 bits, the sign of the source values is not important.

The last three operations are for shifting values. In the case of these instructions the value in the *rs1* indexed register is to be shifted by an amount dictated by the value in the *rs2* indexed register (and the result written to *rd*). Only the bottom 5 bits of the value in *rs2* are used since 31 is the maximum amount that can be shifted. The sign of the *rs1* value does matter, and the first two instructions are for unsigned operations to shift left logical (**sll**) and shift right logical (**srl**). A shift right arithmetic (**sra**) instruction uses a signed value interpretation of the *rs1* value, and will shift right, preserving the sign of the value—that is, the top bit value of the number being shifted, whether 0 or 1, is used to fill the top bits above the shifted value.

Having looked at the R-type arithmetic instructions, there are versions of these exact same instructions using the I-type format. The table below lists these.

name	31:20	24:20	19:15	14:12	11:7	opcode	Description
addi	imm[11:0]		rs1	000	rd	0010011	(rd) = (rs1) + signed(imm)
slti	imm[11:0]		rs1	010	rd	0010011	(rd) = signed(rs1) < signed(imm) ? 1 : 0
stliu	imm[11:0]		rs1	011	rd	0010011	(rd) = unsigned(rs1) < unsigned(imm) ? 1 : 0
xori	imm[11:0]		rs1	100	rd	0010011	(rd) = (rs1) ^ signed(imm)
ori	imm[11:0]		rs1	110	rd	0010011	(rd) = (rs1) | signed(imm)
andi	imm[11:0]		rs1	111	rd	0010011	(rd) = (rs1) & signed(imm)
slli	0000000	imm[4:0]	rs1	001	rd	0010011	(rd) = (rs1) << unsigned(imm)
srli	0000000	imm[4:0]	rs1	101	rd	0010011	(rd) = (rs1) >> unsigned(imm)
srai	0100000	imm[4:0]	rs1	101	rd	0010011	(rd) = signed(rs1) >>> unsigned(imm)

For I-type instructions, the major difference is that, instead of an *rs2* index to fetch a value from a register, the encoded immediate value is used in its place. The instructions have the same name as the R-type instructions but are suffixed with an 'i' (e.g., **addi**). In all other respects these perform the same operations as for the R-type instructions. The eagle eyed amongst you will have noticed that there is no subtract instruction in this list. Since the add immediate instruction is a signed operation, subtraction is performed by just making the immediate value negative.

These few operations, then, are the main data manipulation instructions for RV32I, barring the U-type which we will get to later. Hopefully, you can see that these operations are simple and can easily map these to the kinds of logic you would see in any digital design, with just adds, shifts, comparisons, and logic operations.

Memory Access Instructions

In the first part of the document it was mentioned that a characteristic of RISC architectures was that memory loads and stores were separated from data manipulation, and RISC-V is no exception. The instruction set, for RV32I, provides access to memory for reading and writing bytes (8-bits), half-words (16-bits), and words (32-bits). The tables below show the load and store instructions:

Load Instructions (I-Type)

name	31:20	24:20	19:15	14:12	11:7	opcode	Description
lb	imm[11:0]		rs1	000	rd	0000011	(rd) = signed(mem_byte[(rs1) + signed(imm)])
lh	imm[11:0]		rs1	001	rd	0000011	(rd) = signed(mem_hw[(rs1) + signed(imm)])
lw	imm[11:0]		rs1	010	rd	0000011	(rd) = mem[(rs1) + signed(imm)]
lbu	imm[11:0]		rs1	100	rd	0000011	(rd) = unsigned(mem_byte[(rs1) + signed(imm)])
lhu	imm[11:0]		rs1	101	rd	0000011	(rd) = signed(mem_hw[(rs1) + signed(imm)])

Store Instructions (S-Type)

name	31:20	24:20	19:15	14:12	11:7	opcode	Description
sb	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	mem_byte[(rs1) + signed(imm)] = (rs2)[7:0]
sh	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	mem_hw[(rs1) + signed(imm)] = (rs2)[15:0]
sw	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	mem[(rs1) + signed(imm)] = (rs2)

As stated, there are load instructions for byte (**lb**), half-word (**lhw**) and word (**lw**). An address to read from memory is formed by adding (with signed arithmetic) the immediate value to the value in the register indexed by *rs1*. A memory read is done to this address and the returned value is placed in the register indexed by *rd*. For **lb** and **lhw** the returned value is signed extended to make a 32-bit word (the **lw** read data is already 32 bits). Unsigned versions of **lb** and **lw** are provided, **lbu** and **lhwu**, that do not sign extend the returned value.

The store instructions write bytes, half-words, and words to memory with **sb**, **shw** and **sw**. The address is formed in the same way as for loads, but the value to be written is taken from a second register indexed by *rs2*. Since the destination for a store is memory, there is no *rd* index, and no register values are updated. There are no signed/unsigned differentiated instructions for stores, as the **sb** and **shw** are reducing, not expanding, the values, with the bottom 8 or 16 bits in the *rs2* indexed register used.

PC Update Instructions

For RV32I, the PC updating instructions are classed into two types; those that change the PC on a condition (a branch), or those that unconditionality change the PC (a jump). These have the instruction format types B-type and J-type respectively. The tables below show these instructions:

Branch Instructions (B-Type)

name	31:25	24:20	19:15	14:12	11:7	opcode	Description
beq	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	pc = pc + ((rs1) == (rs2) ? signed(imm) : 4)
bne	imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	pc = pc + ((rs1) != (rs2) ? signed(imm) : 4)
blt	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	pc = pc + ((rs1) < (rs2) ? signed(imm) : 4)
bge	imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	pc = pc + ((rs1) >= (rs2) ? signed(imm) : 4)
bltu	imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	pc = pc + (unsigned((rs1) < (rs2)) ? signed(imm) : 4)
bgeu	imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	pc = pc + (unsigned((rs1) >= (rs2)) ? signed(imm) : 4)

Jump and Link Instructions (J-Type, I-Type)

name	31:25	19:15	14:12	11:7	opcode	Description
jal	imm[20|10:1|11|19:12]			rd	1101111	(rd) = pc + 4, pc = pc + signed(imm)
jalr	imm[11:0]	rs1	000	rd	1100111	(rd) = pc + 4, pc = ((rs1)+ signed (imm)) & ~32'h1

For the branch instructions the values from two registers (indexed by *rs1* and *rs2*) are compared. If the condition is met, then the PC is updated by adding (with signed arithmetic) the instruction's immediate value to the current PC value. If it's not met, then the PC increments as normal (4 is added for RV32I). There are four conditions for branches:

- Equal
- Not equal
- Less than
- Greater than or equal

The instructions **beq**, **bne**, **blt**, and **bge** implement branches on these conditions. Since the sign of a value matters for the last two conditions, there are unsigned versions of **blt** and **bge**—namely **bltu** and **bgeu**.

For the jump instructions, RISC-V defines a jump-and-link approach. That is, the address of the instruction after the jump instruction is saved so that it can be used later on to return flow back to the instruction just after the jump. There are two jump and link instructions, a direct jump instruction (**jal**) and an indirect jump instruction (**jalr**). For the direct jump, the current PC + 4 is saved to the register indexed by *rd*, and the sign extended immediate value is added to the current PC. For the indirect jump, the PC + 4 is saved to *rd* and the sign extended immediate value is added to the value in the register indexed by *rs1*, and the PC set to this result, but with the least significant bit forced to 0 if set (instructions can't start at an odd byte address—but can on a half-word address if compressed instructions supported, hence only the bottom bit forced to 0).

Other Instructions

The last set of instructions for RV32I tidy up some data manipulation and have a small number of 'system' instructions. The tables below show these remaining instructions:

Load upper Instructions (U-Type)

name	31:12	11:7	opcode	Description
lui	imm[31:12]	rd	0110111	(rd) = {imm[31:12], 12'h000}
auipc	imm[31:12]	rd	0010111	(rd) = pc + {imm[31:12], 12'h000}

System Instructions

name	31:28	27:24	23:20	19:15	14:12	11:7	opcode	Description
ecall	000000000000			00000	000	00000	1110011	Service call exception: save pc, pc = exception addr
ebreak	000000000001			00000	000	00000	1110011	Break to debug exception : save pc, pc = exception addr
mret	001100000010			00000	000	00000	1110011	Return from exception. pc = saved pc addr
fence	fm	pred	succ	00000	000	00000	0001111	Fence (barrier) accesses as viewed by other processes

The 'load upper' instructions are really data manipulation instructions, but they use their own special format and are a means to an end, rather than true data manipulating. Since, for the 32-bit RV32I specification, addresses are 32-bits and instructions are also 32-bits, one can't construct a single instruction to load a full 32-bit address into a register using immediate bits. What can be done is to load a large portion of an address into the upper bits (bits 31 down to 12) and then use an **addi** instruction to construct the lower bits (11 down to 0). One could use the shift left immediate instruction (**slli**) but, as this can only do 12 bits at a time, would take three instructions instead of two. There are two flavours of 'load upper' instructions. The 'load upper immediate' (**lui**) simply loads the instruction's immediate value into the upper bits of the destination register, indexed by *rd*, and clears the lower 12 bits. The 'add upper immediate to PC' (**auipc**) instruction constructs upper address bits relative to the current PC value (i.e., the address of the **auipc** instruction). Here *imm*[31:12], concatenated with 12 bits of 0, is added to the PC values and placed in the destination register, indexed by *rd*.

The system instructions are fairly straight forward. The **ecall** and **ebreak** instructions force an exception condition. In the specification we're following, they basically do the same thing. If vectored exceptions (mentioned in Part 1, but more later) are supported and enabled, they will jump to different offsets from the base exception handler address. For reference, the **ecall** would be normally used to break from a running process back into the operation system code for service. The **ebreak** is involved with debugging, where the exception would jump into debugging software—a break point in the code. The **mret** instruction (machine level return from

exception) actually sits outside the RV32I specification and is a privileged instruction. Since we are going to always run in the top-level machine privilege mode, this is a valid instruction, and is vital to exception handling. This instruction updates the PC to the address where an exception occurred, that was saved when the PC was updated to the exception base address (plus any offset). This saved address could have been where, say, an **ebreak** instruction was, or an illegal instruction, or even the instruction that was interrupted by an external event. The address is saved in a CSR register, which is the subject of the next section. So **mret** is called at the end of an exception handler to return flow back to normal.

Lastly, **fence** is used for memory and I/O access ordering. If a system is allowed to reorder memory accesses (for efficiency's sake) then, for certain operations this can cause hazards between different *harts* or other external cores or devices that read and write memory or I/O. The fence instruction forces all memory or I/O accesses that were sent (from this core and hart) to have completed before continuing. For our purposes, with a single hart and core, and no memory sub-system with access re-ordering, it actually will not need to do anything and is a 'no operation'.

This is all the instructions we need for RV32I compliance. It is possible to construct a processor that processes just these instructions but, as I've said before, a practical solution will have control and status registers, specified by the *Zicsr* extension, and this introduces a few more instructions to manipulate these (though, luckily, not many).

Control and Status Registers

In Part 1 of the document, some example register sets were looked at and included among them were control and status registers (CSRs), but in the RISC-V registers we looked at earlier there were no control and status registers. RISC-V does things slightly differently (than processors I've been familiar with).

RISC-V defines a separate 4K word space in which various CSRs are mapped. This space is completely separate from the normal address space access with load and store instructions which can access the CSR registers. New instructions (as part of the *Zicsr* extensions) are required to get to this space. In theory, then, we can have up to 4K 32-bit CSR registers. In the current specification a little over 200 registers are defined but, thankfully, most of these are optional. Before we look at those that are required, the register access instructions need defining.

CSR Access Instructions

The table below shows the six instructions for accessing CSR registers. There are actually only three basic operations, with register and immediate versions of these.

name	31:20	19:15	14:12	11:7	opcode	Description
csrrw	CSR index	rs1	001	rd	1110011	(rd) = csr[index], csr[index] = (rs1)
csrrs	CSR index	rs1	010	rd	1110011	(rd) = csr[index], csr[index] = csr[index] | (rs1)
csrrc	CSR index	rs1	011	rd	1110011	(rd) = csr[index], csr[index] = csr[index] & ~(rs1)
csrrwi	CSR index	uimm[4:0]	101	rd	1110011	(rd) = csr[index], csr[index] = {27'h0, uimm}
csrrsi	CSR index	uimm[4:0]	110	rd	1110011	(rd) = csr[index], csr[index] = csr[index] | {27'h0, uimm}
csrrci	CSR index	uimm[4:0]	111	rd	1110011	(rd) = csr[index], csr[index] = csr[index] & ~{27'h0, uimm}

The three operations are to read-and-write a register, read-and-set a register, or read-and-clear a register. For the register type instructions, the write, set, or clear is done from a value in the general-purpose register indexed by *rs1*. The CSR index field identifies where within the 4K word space the operation is performed, and the *rd* field indexes the general-purpose register where the register's old value will be written. Note that, if the old value is of no interest, *rd* can index *x0*.

The immediate versions of the instructions are similar, but the 5-bit field is unsigned and can, obviously, only affect the bottom five bits of the register being accessed, though many of them have active bits higher than this. This is still useful as many other of the CSRs only have the first few bits defined with sub-fields.

Registers

Of the 200 or so possible control and status registers, only a handful are required. Even within this requirement, some may be defined as fixed at 0. The sub-set we shall look at comes from a real-world example.

In October 2021 Intel released a softcore RISC-V processor as part of its NIOS range for its FPGAs—the NIOS V/m softcore processor. This meets the RV32IA specifications (and implied *Zicsr* extension), where the A extension is for atomic operations (we will discuss standard extensions later). As part of this softcore it has a set of control and status registers as listed in the table below with their names and offsets within the CSR space.

offset	type	name	description
12'h300	read-write	**mstatus**	Machine status register
12'h301	read-write	**misa**	ISA register (indicating which extensions are present—can be 0)
12'h304	read-write	**mie**	Interrupt enable register
12'h305	read-write	**mtvec**	Interrupt/exception vector base address register
12'h341	read-write	**mepc**	Exception program counter (address of interrupted instruction)
12'h342	read-write	**mcause**	Code indicating interrupt/exception that caused a trap
12'h343	read-write	**mtval**	Register holding exception specific information
12'h344	read-write	**mip**	Interrupt pending register
12'hf11	read-only	**mvendorid**	Registered ID of vendor of implementation (can be 0)
12'hf12	read-only	**marchid**	Registered base architecture of the processor hart (can be 0)
12'hf13	read-only	**mimpid**	Implementation ID (revision number—can be 0)
12'hf14	read-only	**mhartid**	HART (hardware thread) ID. Must implement at least 1 HART with ID 0

Two things to note about this table. Firstly the register names in green indicate the registers that *could* legally be set to 0, though this does not imply that they are for the NIOS V/m processor. Secondly, my understanding of the specification is that the **mscratch** register is also requirement, at offset `0x340`.

Concentrating, for a moment, on the registers that are black, these are pretty much all involved with exception handling. Since there is a section on this subject below, I just want to summarise the other registers here, and deal with the exception functionality in the later section.

The **mscratch** register mentioned earlier is just a 32-bit CSR that can be written (or set, or cleared) with a value and has no other function. It has no defined function and is there as a place to hold a useful value for the software. The **misa** register (if not 0) defines which extensions are implemented. Bits 0 to 25 map to the letters A to Z, and when set means that extension is implemented. The NIOS V/m would then have bit 0 (A for atomic) and bit 8 (I for integer) set. It also has two bits (31 down to 30 for RV32I) to define the architecture size (RV32, RV64 or RV128), with `01b` being RV32. These bits can be writable if an implementation can support multiple architecture sizes. For example if an RV64 architecture can be set to operate as RV32. If an implementation can't do this, writing has no effect.

The four 'id' registers are read-only, and (if not 0) give information about the device. The vendor ID (**mvendorid**) is a unique ID, obtained from RISC-V International, for each manufacturer of cores. The **marchid** specifies the microarchitecture. For open-source projects these are also allocated by RISC-V International and have the top bit clear. A vendor may set this number, but the top bit must be set. The **mimpid** register is for vendors to set a version number. The **mhartid** register is a read-only register indicating which *hart* is currently running the code. As has been mentioned before, an implementation can have just 1 *hart*, and its ID must be 0. For multiple

hart devices one of the *harts* must be ID 0 though the others can be any valid number. For our purposes, with 1 hart, **mhartid** will always be 0.

Timer Registers

In the first part of the document, the simplified SoC diagram had a timer that could generate interrupts and it was indicated that this was needed for multi-tasking operating systems to schedule the swapping of processes and threads. The RISC-V specification defines two timer registers but these are deliberately defined outside of the CSR space and are mapped into the normal memory address space. The addresses within the memory map are not defined. The reason for this is that timers often run from real-time clocks and may even be external devices. Running a timer from the core's clock might be possible if it is a fixed clock rate, but many implementations have variable clock rates for power saving when loaded more lightly. The specification doesn't dictate the tick rate for a timer (though it must be possible to derive what it is), but it must be constant and it must be possible to derive real-time from its value.

The timer registers are 64 bits, though for RV32 these are split into high and low registers of 32-bits each and mapped to consecutive addresses. There are only two timer registers; **mtime** and **mtimecmp**. The first counts at the constant tick rate and will wrap if it overflows. The second is a comparison value and will set an interrupt request if its value is less than, or equal to, the **mtime** register. Whether this causes an exception depends on whether it has been enabled or not.

RISC-V Exception Handling

Exception handling is closely tied to the CSR registers. The first functionality required is to enable or disable all interrupts and events (that aren't error conditions). This global interrupt is controlled from a bit in the **mstatus** register (MIE at bit 3). This is the machine interrupt enable bit. There is a supervisor interrupt enable bit (SIE) but we are not supporting this mode. If interrupts are enabled and an event occurs, the MIE bit state is saved to another bit (MPIE at bit 7) and the MIE bit cleared. This is so the interrupt or event does not keep causing a new exception. When the exception code returns (by executing an **mret** instruction) the MPIE value is copied back to MIE. There is one more field to mention in **mstatus**, and that is MPP (machine previous privilege at bits 12 down to 11). This is used if multiple privilege levels supported, to save off (and then restore) which level was active at the point of the exception. For our purposes this is a fixed value for machine privilege (11b). There are other fields in the **mstatus** registers to do with supervisor and user modes which we will not concern ourselves with here and can set to 0 in an implementation.

Having defined a global interrupt enable, there are finer controls of some different classes of exception controlled from the machine interrupt enable register (**mie**). There are both machine and supervisor control bits but we shall limit ourselves to just the machine bits:

- MSIE: bit 3 : software level interrupt enable
- MTIE: bit 7: timer interrupt enable
- MEIE: bit 11: external interrupt enable

The software level interrupts are generated by software writing to some location to raise this type of exception. The mechanism is not specified, but writing to a bit in an implementation specific register could be one way. The timer and external interrupt types are, I hope, self-explanatory. There is an equivalent read-only pending interrupt register (**mip**) with the same bit field definitions that indicates, when set, if any of these three types of interrupts are active.

Mentioned before is that, when an exception occurs, the program counter (PC) is set to some fixed address with, possibly, an offset depending on the nature of the exception. The address that the code will jump to is defined in the machine trap vector base address register (**mtvec**). Since the address will be 32-bit aligned, only bits 31 down to 2 define the address, with bits 1 down to 0 implied as `00b`. Instead, these bottom two bits define whether all exceptions will jump to the defined address (non-vectored, when `00b`) or whether they will jump to an offset from this address depending on the exception type (vectored, when `01b`). The offset is calculated as 4 × the 'cause number' (which we will look at shortly). It has also been mentioned that when an exception is taken, the value of the PC at that point is saved. This is done into the machine exception program counter register (**mepc**), which can also be updated with the CSR instructions. The machine trap value (**mtval**) register may also be updated on taking an exception with implementation specific information to aid an exception handler, or it may be set to 0, and can also be updated with CSR instructions.

The final exception CSR is the **mcause** register. When an exception occurs, and the PC is updated to the exception address, the cause of the exception is noted in the **mcause** register. It is this value that determines the offset taken from the **mtvec** address if vectored exceptions are enabled. The top bit of the address indicates whether it is an interrupt (the bit is set) or and internal exception (bit clear). For the interrupt case only the three classes of interrupt mentioned before (software, timer, and external) are supported, but separate for the machine, supervisor, and user privilege modes. The exception codes just for the machine level exceptions are shown below (with the bit 31 implied as set to 1)

- 3: machine software interrupt
- 7: machine timer interrupt
- 11: machine external interrupt

For the internal exceptions (with top bit at 0), the list of codes is shown below, and is applicable for all privilege modes:

0: Instruction address misaligned
1: Instruction access fault
2: Illegal instruction
3: Breakpoint (**ebreak** instruction)
4: Load address misaligned
5: Load access fault
6: Store address misaligned
7: Store access fault
8: Environment call from U-mode (**ecall** instruction)
9: Environment call from S-mode (**ecall** instruction)
10: *Reserved*
11: Environment call from M-mode (**ecall** instruction)
12: Instruction page fault
13: Load page fault
14: *Reserved for future standard use*
15: Store page fault

Of these codes, 8 and 9 are for user and supervisor privilege modes so we can skip these. The access faults refer to some illegal addressing, for example reading an instruction from memory marked as having no execution permissions. These would normally be flagged by external signalling from the memory sub-system. The page faults would also be flagged externally by the memory sub-system when, say, there is no mapping of a physical page for the virtual address being used (see my article on virtual memory). The access and page faults need not concern anyone until interfacing a core implementation to a memory sub-system.

The ones that are of interest are, firstly, the illegal instruction exception, where either this does not decode to a legal value, or has a sub-field with an illegal value for an instruction etc. Secondly, the address misaligned exception, for instruction, load, and store, will flag if a calculated address was bad, such as a word access being aligned to an odd byte. The breakpoint code is for when the **ebreak** instruction is executed, with the environment call code is for when the **ecall** instruction is executed.

We have now been through everything that is required to put together a logic implementation for an RV32I (+*Zicsr*) compliant processor core. Before we finish

though, for completeness, some other standard extensions should be summarised to show what other features are of interest to processor functionality.

RISC-V Extensions

From the base specification are defined a set of standard extensions to add further functionality. Additional extensions are being defined all the time and added to the specification once ratified, but we will look at some of the original standard extensions to the RV32I base specification. As you may have gleaned, these extensions usually have a letter associated with them, and an implementation is defined by listing the letters of the supported functionality (e.g., the NIOS V/m softcore was RV32IA, meaning it was the base plus atomic instructions). The most common extensions are listed below:

- RV32M adds integer multiplication and division instructions
- RV32A adds atomic instructions to ease multiple processor communication through memory
- RV32F adds single precision floating point instructions
- RV32D adds double precision floating point instructions

The above extensions together, with the base and *Zicsr*, constitute the RV32G specification, or general-purpose processor. RV32G is thus shorthand for RV32IMAFD, and this is considered a useful specification for a general-purpose processor implementation.

Other useful extensions are:

- RV32E drops the regfile registers to just *x0 – x15*, for logic saving
- RV32C adds 16-bit compressed instructions to save coding space in embedded systems

For RV32E, it was realised that to make RISC-V useful for small embedded (hence the E), low-power, low frequency applications it would be useful to reduce the number of registers from 32 to 16 to save on gates. If you refer back to the register set and the ABI names, you will see that in the first sixteen register there is at least one type of register that's in the upper 16 registers, allowing a calling convention to be maintained for the RV32E specification.

The compressed instructions are 16-bit 'abbreviations' of a sub-set of the full 32-bit instructions, and can be mapped one-to-one with a full instruction, though often with restricted parameter values (such as register indexes). The ARM processors have an equivalent with their thumb instruction set, though one difference is that the ARM enters and exits a thumb mode (and does so on 32-bit boundaries) whereas, for

RISC-V, compressed instructions can be freely mixed with the full 32-bit instructions meaning that 32-bit instructions can be aligned on 16-bit boundaries and not just 32-bit boundaries, and this must be handled by an implementation with compressed instruction support.

In all of these descriptions we have restricted the discussion to RV32, but all this is applicable to RV64 as well. In that case all instructions, data sizes and CSRs become 64 bits. All the default instruction behaviour then works on 64-bit values. Because manipulating 32 bits is still useful for a 64-bit implementation, additional data instructions are included, for 32-bit data manipulation, with RV64 over RV32 (though they are not the same opcode as their RV32 equivalents, as these are upgraded to 64-bit operations).

Conclusions

This article has introduced the RISC-V ISA specification. In order to limit the size of the discussion, only the minimum has been reviewed for a compliant implementation; namely, RV32I+*Zicsr*. As well as being an informative introduction to a modern RISC processor, it is also going to be used as a basis for actually implementing a RISC-V core, in logic, to the RV32I specification and detailed discussion of all optional features have been either skipped or only mentioned in passing—though I have attempted to make the article to be a stand-alone introduction to RISC-V for those who do not wish to go the next steps to an implementation.

This article is just a review and does not constitute a formal specification. The specifications themselves are the final word on compliant operation and can be freely sourced on the riscv.org website. The RV32I instructions are defined in volume 1 of the specification and the CSR registers and exception handling operations in volume 2.

The next articles will look at what an RV32I logic implementation might look like as an example, and as a platform for discussing more advanced techniques that could be employed for improved processor performance (though this can't be exhaustive), and also to look at assembly language programming of a RISC-V, as this will be both informative and useful for testing any logic implementation.

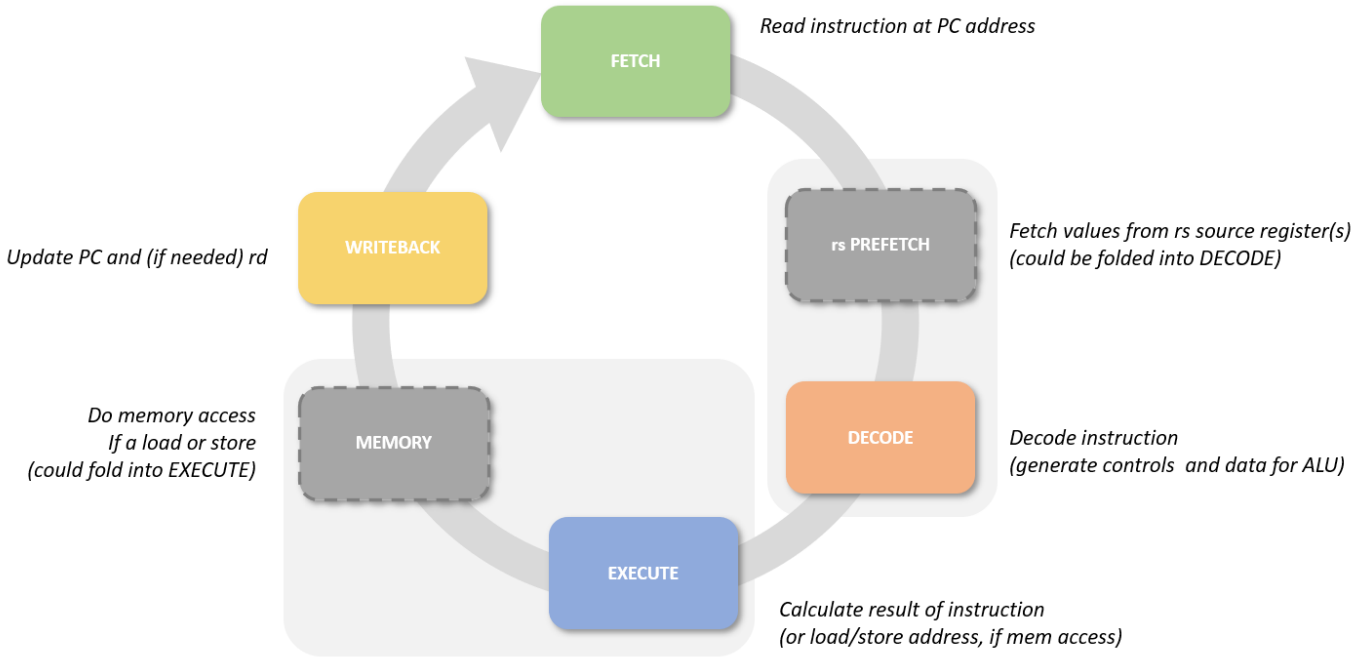
Part 3: Processor Logic

Introduction

In Part 2 of the document, the 32-bit RISC-V base ISA was introduced, along with the *Zicsr* extensions that are really needed for a useful implementation. The specific classes of instructions were defined for data manipulation in and out of internal registers (also defined), memory accesses, and program counter updating. With the addition of control and status registers and exception handling that they mostly control, we should have enough to implement a compliant RISC-V processor in logic.

In this article is discussed approaches to implementation of a processor, and reference will be made to an example open-source Verilog implementation that is available on github for those who want to study the code alongside this discussion. Some of the design decisions will be highlighted and also alternatives and more advanced approaches. The aim of the example implementation is to provide a working design with some common features to make a useful and somewhat efficient core, but in places clarity has taken precedence at the slight cost of speed and logic size. Where I can, these will be 'confessed'. Better open-source implementations exist that will be more efficient in gates, speed or power consumption, or a mixture of these.

Before looking at implementation specifics I want, in the first part of the article, to define what functions the logic will need to perform. There are various operations that are commonly defined for the basic operations of a processor core for executing instructions, with slight variations. The diagram below shows a typical set of these operations.



Starting at the top, without implementation specifics, a processor will fetch an instruction from memory, it will decode the instruction and read the values of the indexed source registers, it will execute the instruction using the register values, it will access memory (if a load or store instruction) and it will write the result back to a destination register (unless a store) or the program counter. The process then repeats, either as the normal flow of execution or the PC was updated due to an exception condition that's occurred, externally or internally.

In the diagram I have shown some optional steps, where the reading of the source registers might be bundled as part of decoding, or possibly split out as a separate prefetch, and for memory accesses combined in the execute step or have its own step. In theory, all of these steps could be done in one big operation and the reality is that different implementations have different sets of combined or split steps to meet specific requirements. None-the-less, the above diagram lends itself to the start of an implementation design, so let's begin with this.

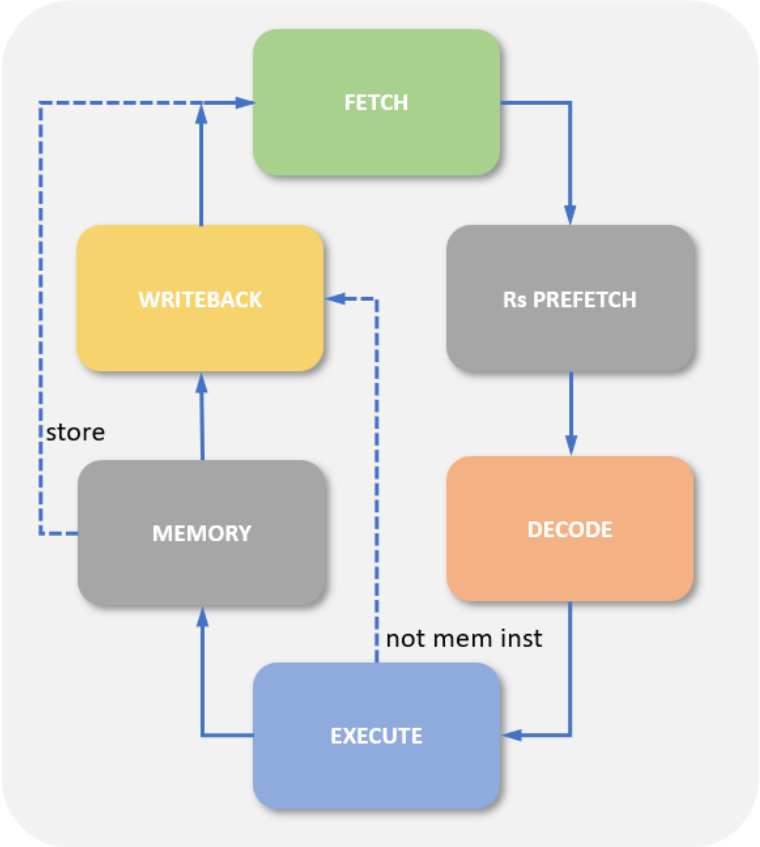
Design Approaches

When designing a solution to a core implementation, as with any design process, a set of conflicting requirements will shape what approach is taken. Perhaps the engineering problem being solved lends itself to a large number of tiny processing elements, so the core design should be as small as possible whilst trading instruction execution speed (see, for instance, the fascinating bit-serial SERV RISC-V core). Alternatively, instruction execution efficiency might be key, and a superscalar and deeper pipeline solution is required (more on these terms later)—the SiFive series 7 processors are dual-issue superscalar, 8-stage pipelined cores. The truth is, it is likely

to be mixture of requirements—power, size, speed, complexity, time to market—and a balance between them all. First, though, let's look at just implementing something that functions. The required functionality from the processor ISA does not change with any of these engineering considerations, and we can keep things simple to begin with.

Finite-State-Machine Based Core

The diagram above, suggesting various operations for an executing processor, looks very much like a finite-state-machine (FSM), if a very simple one. If we were only interested in making a functional core, then the FSM approach is a good strategy. The diagram below adapts the original diagram into a set of matching states of an FSM.



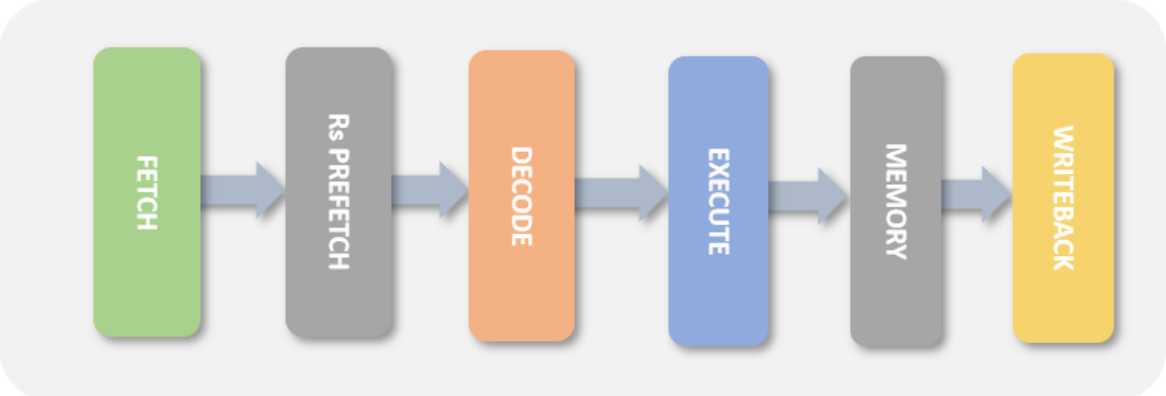
In its simplest form the FSM can just do exactly what the original diagram showed, with one clock cycle for each step (ignoring wait states for now). However, we can already improve on that by skipping states when there are irrelevant for the particular instruction. For example, if the instruction is not a load or store, then the memory state can be skipped. If the instruction is a store, then the writeback step might be skipped—so long as the PC can be updated for any exception conditions in the memory state. Indeed, exceptions that are internal error conditions might occur

in the other stages, and paths from these states directly to writeback when these occur might be another efficiency improvement.

At its simplest, every instruction would take 6 cycles to execute. With some stated improvements, data manipulation and PC updating instructions can be reduced to 5 cycles. This does not sound particularly impressive, but this FSM approach has some major advantages. Firstly, it is very simple to understand and implement and avoids some complex issues of other approaches (as we shall see). It is likely, therefore, to be in a working state much sooner than other methods, with fewer problems along the way. Many early processors are multi-cycle instruction execution implementations, as it was complex enough already to design integrated circuits at that time, and the designers were limited in the amount of logic they could lay down. So, I would suggest that anyone new to this subject and implementing their first processor core starts with this approach. Everything else we are about to talk about does not affect the functional operation of the core and is just there to engineer the efficiency of one or more parameters of an implementation. So, we'll leave the FSM based implementation discussion here, as I hope it is self-evident as an approach to the required logic design.

Pipeline

The classic step of improving the instruction execution speed of a processor core, or indeed any serial data path such as a streaming interface, is to pipeline the design, where each step is a stage in the pipeline feeding into the next step. The diagram below shows the steps 'unwrapped' into a pipeline:



Now, in this arrangement, an instruction is fetched by the fetch 'stage' in a given clock cycle and the returned instruction passed to, in this case, the rs prefetch stage. The prefetch stage, in the cycle after the fetch, can start processing the instruction to retrieve the values in the indexed source registers. In parallel, though, the fetch stage can get the next instruction. As this continues, the pipeline is quickly filled in all the

stages and instructions are being fetched and fed into the pipeline every clock cycle. So, apart from an initial latency before the first instruction is finished, the pipeline completes an instruction every cycle. We have now gone from, at its basic, 6 cycles for the FSM to 1 cycle for a pipeline, and the pipeline doesn't look any more complicated than the FSM, perhaps is even simpler. Of course, there is a price to pay, and we have just created a whole set of problems for ourselves that need solving.

Pipeline Hazards

If a processor only ever did memory accesses and data manipulation instructions, where it was never updating a register that was needed by any of the other instructions in the pipe, then things would be fine. However, it is possible that an instruction's destination register is a source register for an instruction that is in a pipeline stage behind it. Since, during the *rs* prefetch, an old 'stale' value was fetched, the instruction will fail without some intervention. This is known as a register writeback hazard. Fortunately, the internal registers are only updated in one stage, the writeback stage, and it is always just writing to one register (*rd*), so this can be monitored and compared with the *rs* indexes for the instructions at each previous stage. If it matches either of the *rs* indexes, the source value is replaced with that being written to the destination register. This does mean that the indexes for the *rs* values must be forwarded through the pipeline, and the *rd* index and writeback value routed to each stage with a comparator and mux to choose between the value from the previous stage or the writeback value fed-forward.

The second issue we have introduced by going to a pipelined architecture is branch hazards. The default behaviour has been to keep fetching instructions linearly through memory (the default PC + 4 behaviour for the RV32I). If a PC updating instruction is processed then, at writeback, the PC is potentially updated to a new non-linear address and all the instructions in the pipeline behind are then wrong. Dealing with this hazard is known as branch prediction which varies in complexity, from avoiding the problem to quite complex solutions.

Branch Prediction

A whole new article could be written on the subject or branch prediction so we will limit ourselves to a few common solutions. Before we look at these, it should be noted that all these schemes make some assumptions about the nature of the code that runs on a processor. In general, instructions aren't executed from random places in memory but have some localisation, the simplest being located in consecutive addresses in memory. Even when there is a branch, this might be to code that is not

far removed from the current PC address, such as a small local loop. If it were truly random, branch prediction wouldn't really work.

There are two major categories of branch prediction; static and dynamic. The simpler of the two is static branch prediction. Here a predetermined action will be taken when processing a PC updating instruction. Below is a list of three popular strategies.

- Always Take
- Never Take
- Take on negative

The first is to assume that the branch or jump is always taken and to flush the pipeline. When encountering a branch or jump, the pipeline effectively stops fetching new instructions until all the stages have completed and any new PC address has been written to the PC. Whilst flushing, the pipeline is 'stuffed' with no-operations instructions (e.g., `addi x0, x0, 0`), This completely avoids any branch hazard but means execution time is increased to flush the pipe, even if the branch would not have been taken. It also requires a partial decode of the instruction to flag a branch or jump. An improvement might be to pre-calculate the new PC value if taken and start fetching from there. This is okay for branch instructions and jump-and-link (**jal)** instructions, as the potential new PC value is just the current value plus an immediate value from the instruction. The jump-and-link-relative instruction (**jalr**), though, is also a function of a source register, indexed by *rs1*, which might be stale in the register file, with an instruction ahead in the pipe about to update it. Partial decode, addition, and handling stale register values quickly becomes complicated, potentially creating critical path logic or the need to add more pipeline stages to relieve this with the writeback hazards also needing addressing in the new stages.

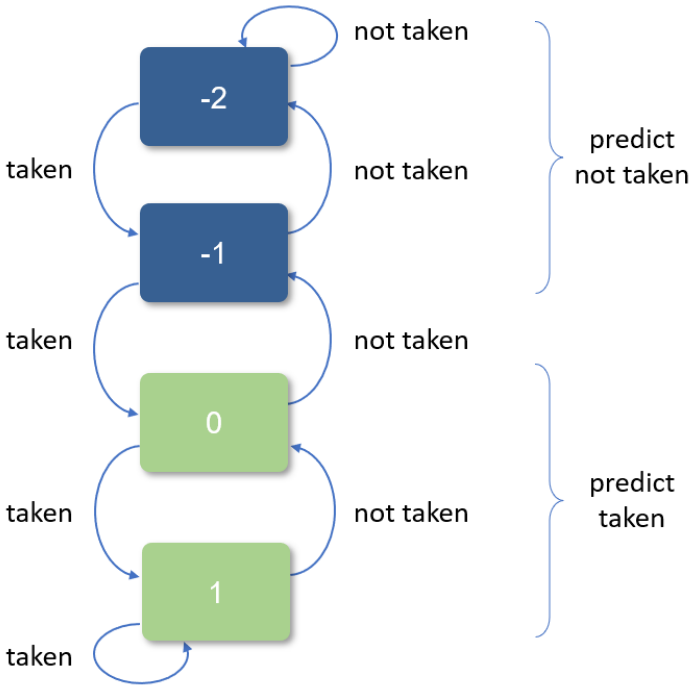
The next strategy is to never take the branch. That is, we will carry on linearly regardless. The advantage of this is that no early partial decode is required and we will be right for a proportion of the time and incur no penalty. How big a proportion depends on the nature of the code running on the core. The disadvantage is that, when wrong, the instructions in the pipe must be cancelled by some method—marked as invalid or forced to a "no-operation" (nop) instruction, etc. Also, if the instruction was a jump, then it could be known in advance that the instruction would be taken. It is perhaps the simplest to implement and is an improvement on the flushing variant of the always-take method but avoids the always-take variant of new PC calculation and complexity of fetching from the non-consecutive address.

The last static method I want to mention is the take-on-negative. That is, for a branch, the branch is taken if the new PC is less-then-or-equal to the current PC (negative) but isn't taken if the new PC is greater than the current PC. To determine

this only the immediate value needs to be processed as either negative or not. For jumps, of course, then it is unconditionally taken. The complexity of logic for always-take methods still applies. The reason this might be an improvement is an assumption about the nature of code—particularly compiled higher level language code—where local loops (such as for-loops of C or C++ for example) are often executed multiple times before exiting, making jumping back more likely than jumping forward. By assuming a backwards branch might be a local loop that is not about to terminate, taking the branch makes the likelihood of being correct higher. Of course, a backwards jump might not be this situation and thus the wrong decision to take.

Lastly, in this section, I want to look at dynamic branch prediction. This differs from static branch prediction as the decision to take or not take is based on state that is going to be kept at run time and so the decision is not-predetermined. There is only space in this article to look at one simple method for illustration purposes, but this is a vast subject and there are many solutions of varying complexity, some of which are proprietary and not in the public domain, I'm sure.

A table is kept of recent branch locations (addresses) along with some state to use in making a decision when next encountered. The table will be finite, so some method for replacing entries when newly encountered branch locations are executed. As such, this is like a cache of branch addresses with a least-recently-used (LRU) replacement algorithm (see my article on caches). Like for static methods this relies on some locality of code execution for some period. The state kept is just a (signed) counter that increments to a maximum value when a branch is taken (that is, actually taken, not the predicted value), and decrements to a minimum value when a branch isn't (actually) taken. For the decision on taking or not taking, if the counter is positive, then the branch is predicted as taken, and if negative it is predicted as not taken. The number of bits in the counter mustn't be so small as to add no real prediction, but not so large that it takes too long to respond to changes in the direction. In fact a two-bit counter is a practical value. The diagram below summarises this:



The starting position can be either at a count of 0 or a count of -1 since no previous information is available so it's a 50:50 situations whether to predict one way or the other. This state is kept separately for each branch address in the table until an old address is swapped out for a newly encountered branch address with initialised state.

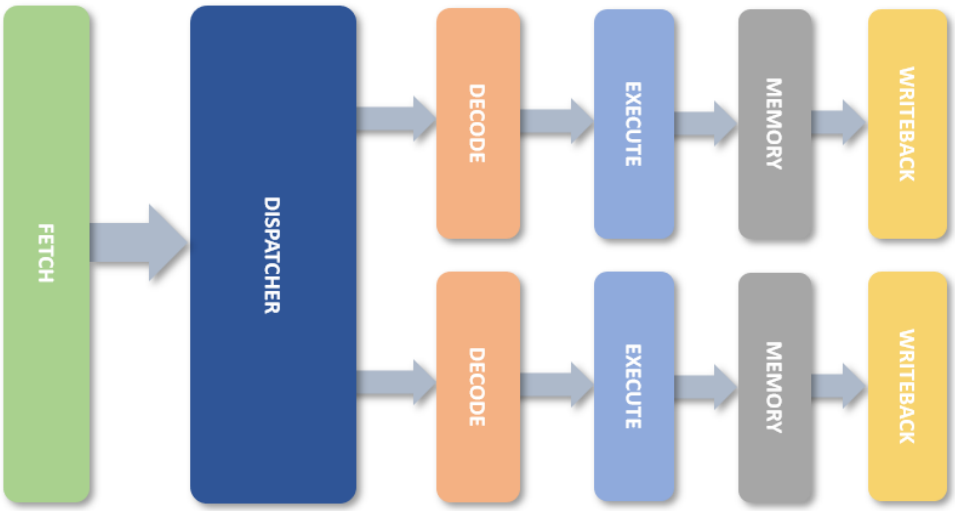
As you can see, branch prediction gets messy pretty quickly, but much attention is given to it as it is all about the efficiency of executing instructions and not pausing in execution due to branching. This is not unlike the added complexity of caches for instruction and data in memory sub-systems which also can stall a pipeline waiting for data to be written or read with wait states.

The memory access wait states are also something that needs to be handled within an implementation which we will look at in the example implementation. This is usually done by stalling the pipeline when wait states are inserted by the memory sub-system. Functionally, this is fairly simple with the update of output state held whilst the memory access is pending. However, care in the design needs to be taken so as not to stall a cycle after a wait state is asserted and can lead to critical paths in the pipeline if using an externally generated signal directly for stalling all stages.

Superscalar

With a pipelined architecture, fixing all the hazards it generates, an efficient branch predictor, and a single cycle memory sub-system, a design could approach running at 1 instruction per cycle. This then is the limit of what we can achieve for a single core, right? Can we do any better? The answer is yes, using superscalar architecture.

The limiting factor is the pipeline, if saturated, can't process instructions any faster. By duplicating the logic within a core, with effectively two (or more) pipelines, instructions can be executed in parallel. The reading of memory instructions needs now to be at a higher bandwidth to match the increased throughput of the core. This might be reading over a wider bus (64-bits when the instructions are 32-bits such as for RV32I) or running the fetch cycles at a higher clock frequency (or a combination of both). The diagram below shows a simplified two pipeline superscalar arrangement.



Two duplicate sets of stages are present, with the increased bandwidth fetch stage, and a new dispatcher stage in between. Like for the initial move to pipelining, going to superscalar creates more problems. If an instruction is to update a register in one pipeline and the next logical instruction has this as a source then if it is dispatched down the other pipeline in the same cycle, then we have a race condition. What if, though, the dispatcher had a following instruction that didn't rely on the result from either of the instructions in front and didn't write back to any register that was a source for them? Then the dispatcher could send that instruction into the other pipeline for execution, "out of order". The instruction that was paused can then be sent into the pipeline for execution. This keeps the pipeline full without changing the result. For example, if the first instruction is `addi, x1, x0`, 1 (put 1 into *x1*) followed by `addi x2, x1, 1` (add 1 to *x1* and place in *x2*), then `addi x3, x0, 6` (place 6 into *x3*), then the last instruction can be executed in parallel with the first, then the second instruction dispatched. The result will still be *x1* = 1, *x2* = 2, and *x3* = 6.

Here I have assumed that the pipeline is executing at 100% but, of course, memory accesses can stall a pipeline, branch prediction may fail and cause flushing etc. The dispatcher will actually keep tabs on all the stages for all the pipelines about which destination registers are to be updated. It will prefetch later instructions, up to the depth of the pipeline, in order to lookahead about whether any instruction is a

candidate for dispatching out-of-order so it can keep the pipelines as full as possible. Of course this is all complicated by branch and jump instructions—and you thought branch prediction was messy. We are getting to quite advanced features now, and still are only just looking under the hood, but we must stop here and look at an example implementation.

Example Implementation

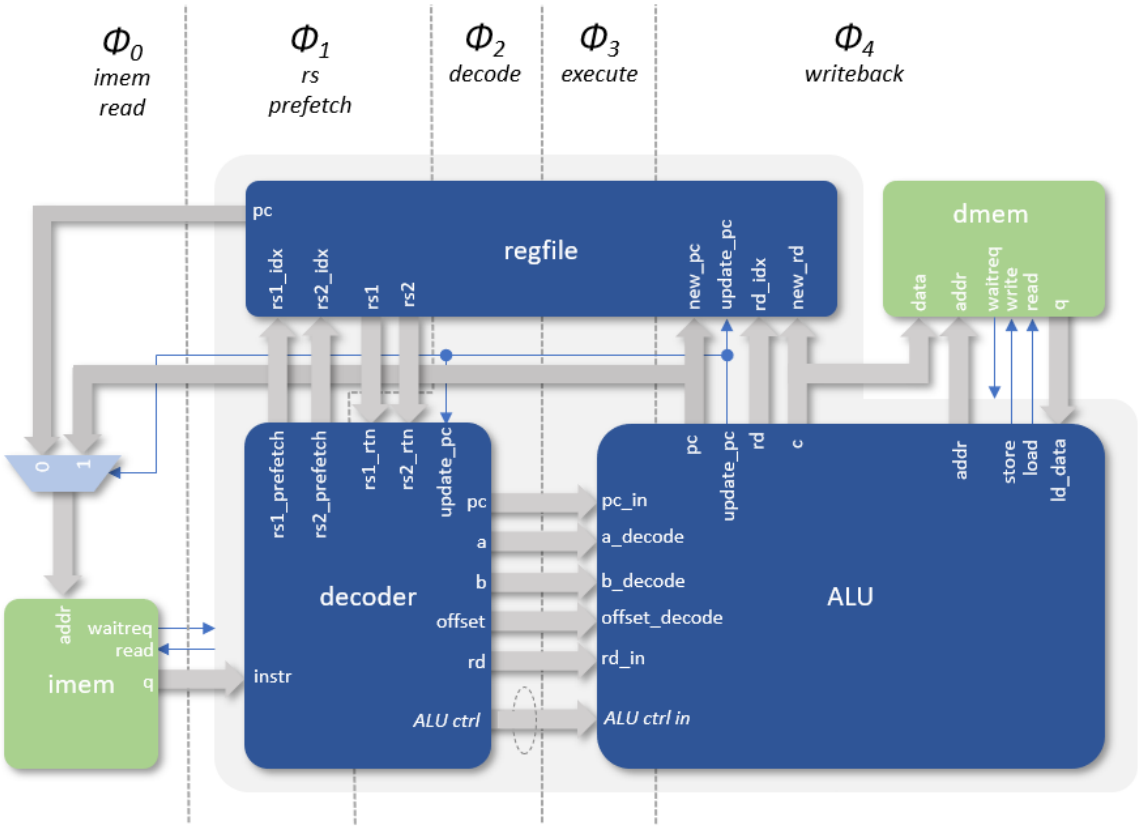
To finish this article, I want to review the design of the RISC-V implementation on github to illustrate some of the topics discussed above. This will not be exhaustive as much of the detail of the design is given in the logic's documentation. Not everything we've talked about is implemented to keep things simple enough to understand but to have advanced enough features for an efficient design. The implementation has the following specification.

- RV32I + *Zicsr* + RV32M, configurable for RV32E
- 5-stage pipeline
- 1 cycle for data manipulations and stores, 3 cycles for loads (plus wait states), 3 cycles for branches when taken (else 1 cycle)
- "Never take" static branch prediction policy
- Single *hart*

The RV32M and RV32E features, though available, need not bother us here and can actually be configured out.

Top Level

The base RV32I logic is implemented as a 5-stage pipe in 3 separate blocks. A register file (regfile) implements the internal registers and the program counter. As such it spans both the fetch stage, by providing the PC, and the writeback phase where registers and the PC are updated. The rs prefetch and decode stages are implemented in the decoder block, and the execute and writeback phases are implemented in the ALU block (with regfile register updates in the writeback phase as well). The diagram shows the top-level block diagram arrangement.



The diagram does not show the *Zicsr* or RV32M blocks. These logically sit in the execute phase and, in this implementation, they are in separate modules. This is so that the core can be easily configured without these features for reduced size if those functions are not required. It requires some additional muxing to do this but appeared not to be on the critical timing path. Otherwise, the additional instructions could have been added to the ALU and the CSR registers to the regfile blocks. The decoder already partially decodes these instructions and forwards them to the blocks.

Register File

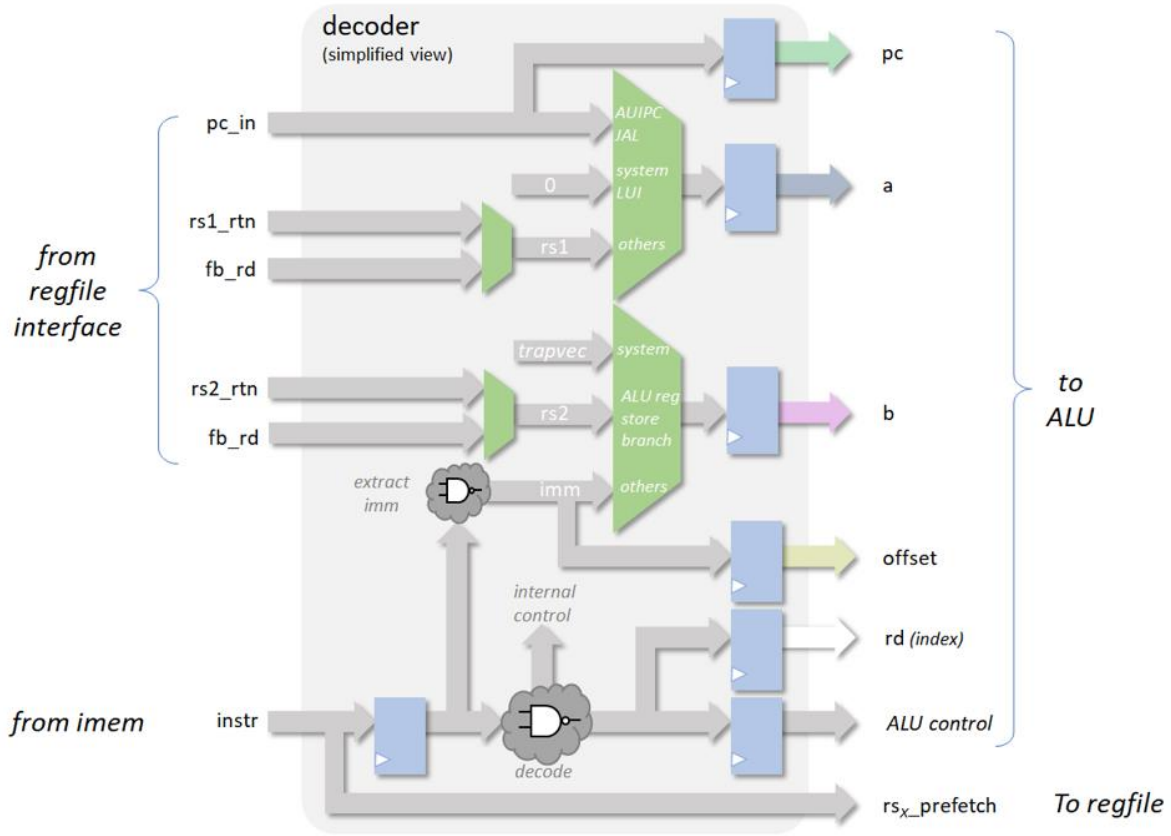
The regfile block contains the general-purpose registers and PC. It is configurable to use either logic gates or memory. Which is best depends on the target technology. FPGAs tend to have available memory blocks which, if not all already allocated, can be used for the registers, saving on consuming logic resources which might be at a premium in a small device. There is an issue, though, if the memories do not have dual read ports. R-Type instructions, as discussed in Part 2 of this document, have two source register to fetch. Either an additional pipeline stage for fetching the registers separately can be used (not preferred) or, if resources allow, two memories can be used. When updating registers during writeback, both memories are updated with the same value at the same offset. When fetching source registers data, one memory is allocated for *rs1* and the other for *rs2*. For targeting an ASIC, whether

logic or memory is more efficient will depend on the technology. A dual port memory is more likely to be available for ASICs though.

Whether a memory or logic, the registers themselves are just 31 registers of 32 bits wide. Logic intercepts reads and writes to *x0,* so it always reads 0. The regfile will increment the PC by default, unless the writeback stage indicates an update.

Decoder

The decoder block extracts the source register indexes and sends then to the regfile for reading the value during the rs prefetch phase. This is done for *rs1* and *rs2* regardless of whether the instruction ultimately require them. The instruction is then registered for the decode phase. The diagram below shows the main functions within the decoder logic:

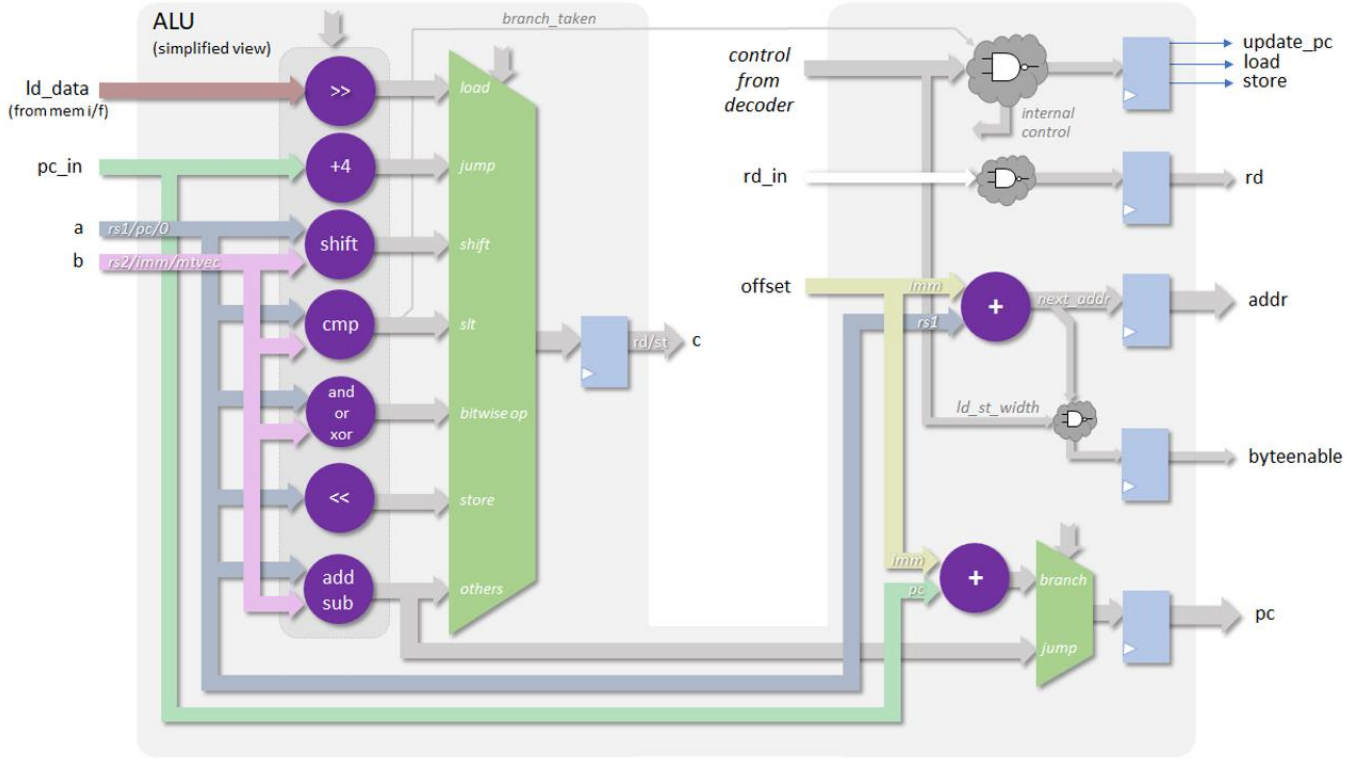


The decoder extracts the immediate bits from the instruction. Since these bits vary according to instruction type, each type is extracted and then the correct one selected and, where applicable, sign-extended, within the decode phase. The register values returned from the prefetch phase are routed to one of two decoder output parameters, *a* and *b*, for routing to the ALU. This diagram shows that these returned register values can be overridden by feedback *rd* values from the writeback stage, discussed later. As well as *rs1* for *a*, and *rs2* for *b*, these outputs can be selected to be

the PC value (from regfile) or 0 for *a*, and the trap vector (Zicsr's **mtvec** + offset value) and immediate value for *b*, depending on the instruction. The diagram shows which type of instructions these are selected for. Other decoder outputs for the ALU include the PC value forwarded, the immediate value to be used as a PC offset, the instruction's *rd* index value and the set of controls to select the ALU execution, decoded from the instruction. These are essentially a bitmap of what operations to perform in the execute phase. All these outputs are registered to place them in the execute phase.

ALU

The ALU is essentially going to take the *a* and *b* parameters from the decoder and generate a result on a *c* output. This is true for all the data manipulation instructions and the store instruction, whilst PC and memory read data are used for loads and jumps respectively. Some addition outputs are generated for memory accesses and PC updates. The diagram below shows a simplified block diagram for the ALU.



The controls from the decoder select which operation to perform. The load instruction has an alignment operation on data returned from memory and the jump instruction takes the forwarded PC and adds 4. The rest of the instructions use *a* and *b*, as selected during the decode phase. The result is registered as output *c*.

In addition to this main ALU operation, the block generates a strobe for updating the pc, with the new PC value being either the PC input plus the forwarded offset value

for branches, or *a* plus *b* for jumps (which contain PC and immediate values). Load and store strobes are generated as appropriate, with an address output calculated from the forwarded offset value and the *b* parameter. From this address any byte enables are calculated from the lower bits. The value to be written for stores is on the *c* output. Finally, the *rd* index value is output for the writeback of the *c* result to the register file.

In the Verilog implementation the ALU code is not fully optimized for gate count. For example, there are three shift RV32I shift instructions, **sll**, **srl** and **sra**. In the code this is handled by calculating the result for each of the possible instructions and then selecting the required value depending on the particular instruction active. The code snippet below shows this:

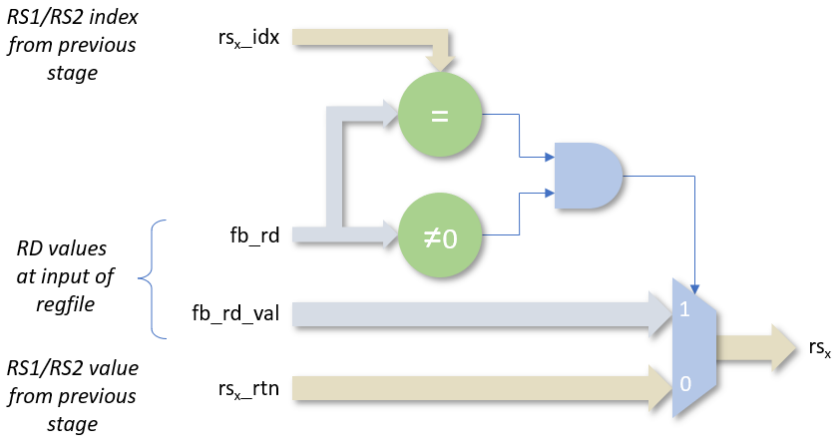
```
wire signed [31:0] a_signed    = a;
wire        [31:0] sll         = a        <<  b[4:0];
wire        [31:0] srl         = a        >>  b[4:0];
wire        [31:0] sra         = a_signed >>> b[4:0];

wire        [31:0] shift       = ({32{shift_left}}                & sll) |
                                 ({32{shift_right & ~shift_arith}} & srl) |
                                 ({32{shift_right &  shift_arith}} & sra);
```

Hopefully it is clear how this operates to produce a shift result, but the three innocuous lines with the shifts hide a lot of synthesized logic, with 32-bit barrel shifters which use a fair amount of gates. A single signed barrel shifter might have been employed with `a_signed` a 33-bit sign extended version of a. For right shifts, `a_signed` is then used for both **srl** and **sra**. For shift left, the input of the barrel shifter is `a_signed` with bits 31:0 bit reversed and bit 32 set to 0. The output of the barrel shifter is then extracted and bits 31:0 bit reversed for the result. This would result in a single 33-bit barrel shifter and some additional muxing, which ought to consume less resources, but may add to the timing path. If timing is not critical then this is a useful optimisation, but the code is less clear. Actually, even the shift for aligning data on loads and stores could also use the same barrel shifter, adding additional muxing to the inputs and outputs. Similarly, a maximum of 2 adders/subtractors are needed (**jal** and **jalr** add 4 to the PC, plus the new PC value addition calculation), but actually 4 are used for adds, subtracts, new PC calculation, and PC + 4. In the implementation, clarity and speed where chosen over resources. All these decisions are a classic compromise between clarity, logic resources and speed.

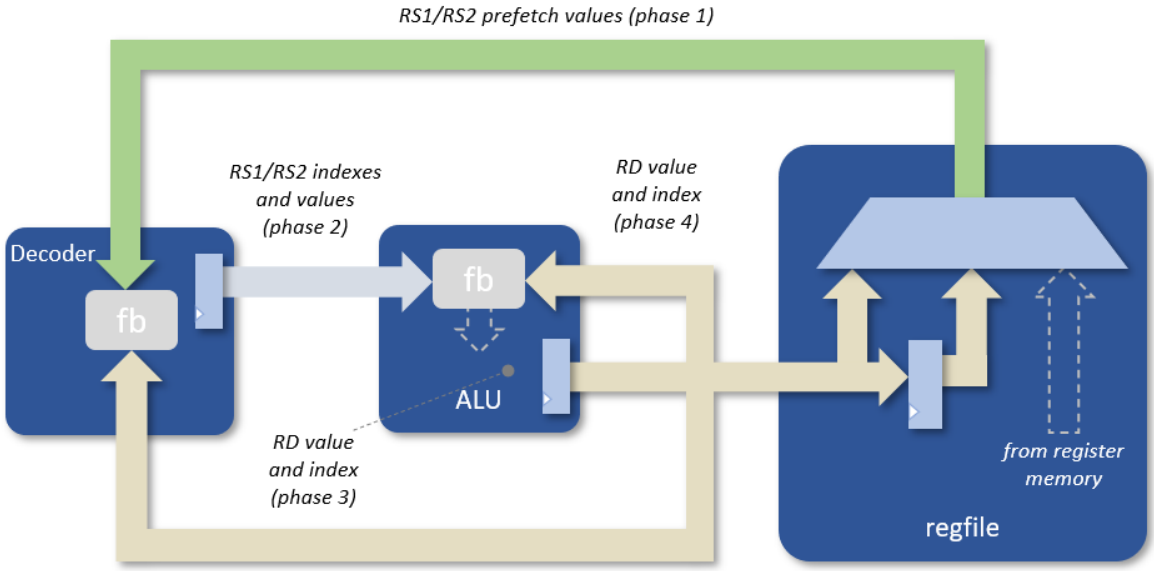
Hazards and Stalls

In the logic implementation, write back hazards are dealt with in the decoder and ALU by comparing the *rd* index on an active writeback with the *rs* indexes for that

stage. The diagram below shows the logic for selecting between the input from the previous stage and the fed-back *rd* value. Note that the writeback logic ensures *rd* is 0 when no active writeback.



Within the register file, if using memory, it takes two cycles before a value written to the register array can be available at the output. When the decoder reads from the registers, if currently writing to a matching *rs* index then the write value is sent back. If matching against the value in the previous cycle, this is selected, otherwise that from the register file is sent. This is illustrated in the diagram below:



To deal with branch hazards under the chosen never-take policy, whenever a branch (or jump) updates the PC, the ALU cancels the next instruction it was about to execute and forwards this 'branch taken' state to the decoder to clear its output state, effectively turning the decoder output to a "no-operation" so the ALU will not do anything with this instruction as well.

Stalling on memory accesses is a matter of detecting a wait state and forwarding a stall signal to all the stages to hold their state. Global stalls can be a source of critical path and faster means might be employed such as a shallow FIFO between each stage where an output is held-off when the FIFO is full. This keeps the stalling local to an interface between two stages, rather than global, but is more complicated to implement and costly in gates—the outputs from the some of the stages can have many bits.

Zicsr

The control and status registers are implemented in a separate module and are logic gates rather than an array. As indicated in Part 2, only a small sub-set of the possible registers will be implemented, similar to the NIOS V/m, with just a few extra. I won't go over their detailed function which is discussed in the last article but is mainly concerned with exception handling, with some additional counters.

The *Zicsr* logic handles both the external exceptions (interrupts, software, and timer) and the internal exceptions. The interrupt and software exceptions are signals that originate from outside the core, whereas the timer interrupts are part of the *Zicsr* logic. The timer can be read and updated from outside the core so that it can be mapped into the memory address space, as per the specification. The timer can be configured to be a 1µs real-time count, or just reflect the clock cycle counts in **mcycle** and **mcycleh**, which are always implemented. Since this implementation has a fixed clock, using the cycle counts is valid. If it ever had a variable clock input, that would not be true.

All the internal exceptions originate from the decoder, either as decoding an **ecall** or **ebreak** instruction, or as an error condition, such as misaligned addresses etc.

The *Zicsr* block has outputs to update the PC on an exception and also to write to a general-purpose register when executing a CSR instruction. The decoder partially decodes the CSR instructions and forwards the parameters to the *Zicsr* block to complete decoding and access the registers. The system instruction, **mret**, is decoded by the decoder block, and a flag sent to the *Zicsr* block. In this case the decoder sends a "no-operation" to the ALU as the *Zicsr* block will handle the instruction. The logic updates the CSR register state and also updates the PC to the exception return address that saved in the **mepc** register when an exception was generated.

The Zicsr logic also, configurably, adds the retired instruction counter registers (**minstret** and **minstreth**) which are incremented on a pulse from the ALU for each instruction completed.

Conclusions

In this article we have looked at what the minimum logic might be to implement a RISC-V processor, starting with just a functional approach based around an FSM. From then on, we looked at ways of improving on this basic design, not to add functionality but to make it operate more efficiently.

Starting with pipelining, the design could suddenly process more instructions per cycle, but at a cost of introducing some hazards which needed to be solved with writeback feedforwards and branch hazard handling. The branch hazards were dealt with using branch prediction algorithms of varying complexity and efficiency and were based on some assumptions about the nature of executing code. This got quickly messy and is only the beginning of the range of possible algorithms employed. With these methods the design could approach executing (in the limit) one instruction per cycle. Looking at superscalar architectures, this can be improved even further but, once again, this has added complications. Complex logic is needed for multiple pipelines and out-of-order execution of instructions without altering the functional results.

Finally, we reviewed an implementation that sits somewhere between the simplest FSM based approach and the complex solutions already looked at which, hopefully, can give a useful example of an IP core for further exploration. The review was cursory, and the implementation documentation gives a fuller picture of the design details for those interested in following this up. Perhaps have a go at implementing a core yourself.

In the next, and final, article we look at the RISC-V assembly language so that the core can be programmed. This may not be of interest to everyone reading these articles, and so it has been left until last, but it ought to be informative and for those wishing to run simulations on the example core (or its accompanying instruction set simulator) will allow them to explore executing all the different instructions.

Part 4: Assembly Language

Introduction

In this, the last of the four parts on processor design, I want to look at assembly language programming—at least give an introduction and overview so that an individual can start writing basic programs. Now many people might say that this is not so relevant these days, and only a specialist subject, as it is likely that any software being written for a processor to run will use a higher-level language from C through to Java and everything in between. Whilst that may be true to a large extent, I think it will be useful to know how assembly language works and to take the first steps from this place to the more machine independent higher level languages. After all, a compiler just converts the language being used into a set of assembled processor instructions.

In this document we will, of course, use as a case study the RISC-V assembly language with the gnu toolchain so that this directly follows on from the previous part of the document. Links will be provided in a later section for those that wish to follow up on this. There will also be links to an RISC-V instruction set simulator so that assembled code can be experimented with and run without the need to have a RISC-V hardware setup.

In the following sections the instruction format will be discussed, mapping directly to the RISC-V instructions, before going through the layout of an assembly language program and its sections. Then compiling the code to a program that can be actually run and getting additional information from this process will be looked at. Executable code can also be disassembled to view the assembled code it's made from, and we will see how the toolchain tool can do this for us. To finish I want to show how to run an executable constructed from assembly language on the ISS so interested people can see what's going on and how instructions alter the processor state. The provided ISS does have a remote gdb interface so that those who know how to drive this can step through code. There is no room in this part for details but the documentation with the ISS details how this is achieved.

Assembly Language

I have discussed before that the fundamental code that runs on a processor is machine code. That is, the raw binary values that make up the instructions, as we saw in the second part of this document. In the early days of computing, this was how a program was labouriously entered into a computer, either directly or by creating paper tape with the codes entered on them. I, myself, remember entering machine

code into some programable test equipment by setting switches to on or off and then pressing a load button, for a set of instructions listed in a test manual. If you got any step wrong, you had to reset and start again.

Assembly language was the next step where the instructions were symbolically represented in a source file, and then "assembled" into the machine code instructions using an "assembler" program. The symbols used are just the instruction names and registers that were documented in part 2 for RISC-V. So instead of `0x05d00893` we can write `addi x17, x0, 93`. I defy anyone to claim that the former is the better choice. So, let's look at how all the RISC-V instructions previously discussed can be represented in assembly language.

Instruction Formats

In part 2 of the document, the six instructions formats used for the base instructions of the RISC-V ISA are written in particular ways in assembler code. For the most part these are the natural way one might write these.

The R-Type instructions for register-to-register data manipulation have the following general format: `<op> rd, rs1, rs2`, where op is the instruction. Some examples:

- `add x3, x1, x2`
- `sla gp, ra, sp`

Note that the first example uses the *x* register index format, whilst the second uses the ABI names which are, if you recall, aliases for the registers. Both methods are valid. The I-Type format for immediate forms of the data manipulation instructions is very similar in format to R-Type: `<op> rd, rs1, imm`. Here *rs2* is replaced with an immediate value. Some examples:

- `ori  x3, x1, 0x123`
- `stli gp, ra, -1`

The `load` instruction is also an I-type instruction but has a different format, namely: `<op> imm(rs1)`. This indicates that the *rs1* register's value is an address, modified by the immediate value (i.e., added to it—noting that `imm` might be negative). The data is read from that address and stored in the destination register. Some examples:

- `lw x3, 64(x1)`
- `lb ra, -4(sp)`

Similarly, the store instruction (S-type) uses parentheses to indicate a memory access, and has the format: `<op> rs2, imm(rs1)`. Again, the *rs1* register value is the

address for the store, offset by the immediate value, and *rs2* indexes the register with the data to be stored. Examples:

- `sw x6, 16(x19)`
- `sh s1, -14(t0)`

With branch instructions (B-type) we're back to a format somewhat similar to I-Type: `<op> rs1, rs2, imm`. Here *rs1* and `rs2` are compared (there's no destination register) and the immediate value is used to alter the PC relative to its current value. Some examples:

- `bge x12, x31, -16`
- `bne gp, t6, fail`

Note that the second instruction doesn't have a numeric value. This is a "label". We will see shortly that, in assembler, one can put labels next to bits of code and then reference them. The assembler will work out what value to put for the immediate value when the code is assembled to go to that place in the code next to the label. The jump instruction **jal** (J-Type) also use an immediate offset, though more bits are available for larger jumps, but the assembly format is still the same and labels can be used: `<op> rd, imm`. For example:

- `jal ra, subroutine`

The companion jump instruction, **jalr**, is actually an I-Type format, for example: `jalr x1, x2, 2000`. Finally, the load-upper-immediate instructions (U-Type) are laid out like J-Type but, again, the number of immediate bits is different, though in assembler one does not need to worry about this except for the range of legal values that can be used. E.g., `lui 0x80000`.

Example Code

Now we know how to put down individual instructions in this symbolic form, we are ready to write a program. A short example will serve to see how a program is constructed and some additional details of an assembly language program will be introduced and discussed. The diagram below shows a simple program.

```
1   #
2   # Example RISC-V assembly program
3   #
4           # ----------------------------------------
5           # Program section (known as text)
6           # ----------------------------------------
7           .text
8
9   # Start symbol (must be present), exported as a global symbol.
10  _start: .global _start
11
12  # Export main as a global symbol
13          .global main
14
15  # Label for entry point of test code
16  main:
17          ### TEST CODE STARTS HERE ###
18
19          addi  x1, x0, 12    # Add 12 to x0 (= 0) and put in x1
20          addi  x3, x0, 0     # Make sure x3 is 0
21          lui   x3, 0x1       # Load x3 upper bits (31:12) with 1 (= address of 0x1000)
22          lw    x3, 0(x3)     # Load a word from memory at byte address labelled data
23          xori  x3, x3, -1    # XOR x3 with -1 (0xffffffff) and put result back in x3
24          addi  x4, x0, 0     # Make sure x4 is 0
25          lui   x4, 0x1       # Load x4 upper bits (31:12) with 1 (= address of 0x1000)
26          sw    x3, 4(x4)     # Store x3 to memory at byte address in x4 (0x1000 + start of data), offset by 4
27
28          ###    END OF TEST CODE    ###
29
30          # Exit test using RISC-V International's riscv-tests pass/fail criteria
31          li    a0, 0         # set a0 (x10) to 0 to indicate a pass code
32          li    a7, 93        # set a7 (x17) to 93 (5dh) to indicate reached the end of the test
33          ebreak
34
35          # ----------------------------------------
36          # Data section Note starts at 0x1000, as
37          # set by DATAADDR variable in rv_asm.bat.
38          # ----------------------------------------
39          .data
40
41  # Label for the beginning of data
42  data:
43          .word 0x12345678    # Test word data value
44
```

In general, you can see the instructions in the middle of the example surrounded by some additional paraphernalia. At the top of the example are lines starting with '#'. These are for comments where anything after the '#' symbol is ignored by the assembler program, thus allowing ordinary text to be added to give information about the program by the implementer. The first 'active' line of the program is "`.text`" on line 7. This is a section marker declaring, in this case, that what follows is an actual program—known as text in assembly language parlance. Programs and data are usually separated into different areas memory. For example, in an embedded system, the code might be in a ROM whilst the data is stored and retrieved via a RAM. These will be mapped into the address space at different offsets, so in the code we mark program sections with `.text` and data sections with `.data` (which can be seen on line 39). When assembled, areas marked as text will be put in the program space, and areas marked data will be put in the data space. The assembler is usually told where the start of these sections are or will use a default. Indeed, line 32 loads the address `0x1000` to *x3* on the assumption that this is where the data section starts. Later we will see how to use the label so that this isn't necessary. A program can switch back and forth between sections in the source code if desired, simply marking the beginning of a section appropriately. There are various

other types of section. A couple of other common sections are `.rodata`, which is the same as data but can't be written to, so may be part of a ROM. There is also a `.bss` section, which means "block starting symbol" for historic reasons but means an area of reserved unitialised memory. Where the `.data` section defines memory with some initial values, the `.bss` section defines an area that is known at compile time needs to be reserved but has no set initial value.

After the `.text` line, at line 10 is a label called `_start`. Labels were mentioned as part of the examples for the instructions. A label is added as a name which must start with a letter or certain symbols (not numbers though), followed by a colon (:). The address of the first instruction after a label will be inherited by that label and thus the label is an alias for that address. The `_start` label is special though, as it must be present somewhere in the program once, and only once, (or programs if multiple source files) and is where the program will start executing from.

Immediately after the _start label there is a '`.global _start`'. This makes the label available externally to separately compiled programs and compiler and linker programs. The default is that a label is only visible within the program file in which it is defined. However, one can use the `.local` directive explicitly to say a label is local only. Note that the `.global` directive can be on a separate line if desired. Indeed, on line 13 and 16 the `.global` directive for a label `main` is on a separate line and before the label declaration. This is all legal.

We then have the program itself using the symbolic notation outlined in the last section. This is a simple linear set of instructions (no branches or jumps in this example) that terminates with an **ebreak** instruction. The comments describe what effect each instruction has.

Other Directives

After the program instructions, the data section is declared (`.data`) and a label added at the beginning called `data`. This has a single word added to the section as `.word 0x12345678`, placing that value in the first four bytes of the data section. The `.word` defines a 32 bit value, and there is also `.byte`, `.half`, and `.quad`. for different sized words. A comma separated list of values after the directive will place these in consecutive order in memory. In addition, a `.string` directive will store the bytes of a string in the data memory terminated with a zero, and a `.zero` directive, followed by an integer count, will place that many zeros in memory.

In addition to the directives mentioned above in relation to the example, some additional useful ones are worth mentioning here. The `.equ` directive allows an alias to be defined for a value. E.g., `.equ INVERT, -1`. The example code uses literal

values throughout the code, but if the above were to be defined at the top of the file, then the -1 of line 23 could be substituted with `INVERT`, giving a clearer indication of the code's intent. These, then, are like labels, aliasing constant data values rather than address locations.

Another useful set of directives are for alignment. As the assembler constructs code, it keeps tabs on the address location, incrementing for each instruction. It may be that a new section of code or data needs to start on some boundary that is aligned to, say, a page, a double-word or whatever. To enforce this the `.align` and `.balign` directives are used. As a minimum they are followed by a byte count for the alignment (in powers of 2). So `.align 8` will move the location counter to the next address that falls on an 8 byte boundary (i.e., one that has the bottom 3 bits as 0). The alignment directives can have a second argument that sets the value of the padding bytes. E.g., `.align 8, 0xff` would set any padded bytes to `0xff`. The default is `0x00` if there is no second argument. The reason there are two types of alignment directive is that `.align` can be different for other processors, specifying the number of low order bits to be 0 instead of the alignment bytes. The `.balign` is unambiguous and always specifies the byte alignment (and there is a `.p2align` for specifying alignment using zeroed low bits).

These directives discussed here are not exhaustive, and the full list of those supported in the RISC-V toolchain are documented in its assembly programmer's manual.

Macros

One last couple of directives I want to talk about are `.macro`/`.endm`. These form a pair to define a section of code that can be inserted in a program. It is possible, when writing code, that one finds oneself writing very similar code in various places with just perhaps the values being used that are different. A macro can be used to define a set of operations that can then be placed in an assembler program and substituted with the defined body of the macro. The macro definition can have arguments so that the code generated is modifiable when instantiated. An example:

```
.macro INITBYTE3REGS reg1, reg2, reg3, bval
  addi \reg1, x0, bval
  addi \reg2, x0, bval
  addi \reg3, x0, bval
.endm
```

In the code the above macro can be used anywhere to insert instructions:

.

```
    .
    .
xori x20, 0x75
INITBYTE3REGS x1, x3, x17, 0xaa
lui  x3, 0x2000
    .
    .
    .
```

In this simple example the macro is defined to set a byte value in three registers. Which registers are not specified in the macro but use arguments for defining them when the macro is used, along with an argument of the byte value. The use of the backslash allows the argument to be reference in the macro body—e.g., `\reg1`. In the code, the macro can be instantiated with arguments, filled in to define the registers to be updated and the value to be set in each by substituting the backslash argument references in the macro definition with those provided. In this usage of macros, it is like defining higher level operations, constructed from the simpler instructions. Note that this is different from calling a subroutine (a section of code that's called using, say, **jal**, and then returned from using the saved return address). In that case there is one copy of the code in memory, and arguments would be passed by setting predetermined registers with the argument values. Using a macro places a copy of the code at each instantiation, effectively the macro is expanded to instructions at that location. When disassembling, the individual instructions would be seen for that part of the code.

Macros, then, are just expanded at assembly time and, therefore, can have other contents such as directives and definitions and need not be restricted to sections of instructions. In fact, much more is possible in assembly language than introduced here and I'd encourage you to explore further and try programming yourself.

Pseudo-instructions

Another feature of an assembler is that it can alias instructions to have more meaningful names. In this and the previous parts I have mentioned that the instruction `addi x0, x0, 0` is a no-operation. It is not necessarily obvious in some code that this instruction is meant to be a no-operation at first glance. It might be better if the defined instructions set had a separate `nop` instruction. Well, the assembler has a set of pseudo-instructions to alias the processor base instructions into more meaningful names. Below is a sub-set table of pseudo-instructions for the RV32I instructions set.

Pseudo-instruction	Base instruction	Description
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`jalr rs`	`jalr x1, 0(rs)`	Jump and link register
`ret`	`jalr x0, 0(x1)`	Return from subroutine

I won't list all the pseudo-instructions here as it would get too long, but there is a set for the *Zicsr* control and status register instructions, such as `csrw csr, rs` which maps to `csrrw x0, csr, rs`. There are also some that map to two instructions. A couple of examples are shown below:

- `call offset` maps to:

```
auipc x1, offset[31:12] + offset[11]
jalr  x1, offset[11:0](x1)
```

- `la rd, symbol` maps to (when not position independent code):

```
auipc rd, delta[31:12]+delta[11] # where delta = symbol – pc
addi  rd, rd, delta[11:0]
```

With this all in mind, we can re-write the example code from before using pseudo-instructions and also use the label for the data to make the code independent of where that might be located.

```
1  #
2  # Example RISC-V assembly program
3  #
4          # ----------------------------------------
5          # Program section (known as text)
6          # ----------------------------------------
7          .text
8
9  # Start symbol (must be present), exported as a global symbol.
10 _start: .global _start
11
12 # Export main as a global symbol
13         .global main
14
15 # Label for entry point of test code
16 main:
17         ### TEST CODE STARTS HERE ###
18
19         addi  x1, x0, 12     # Add 12 to x0 (= 0) and put in x1
20         la    x3, data       # Load x3 with address of data label
21         lw    x3, 0(x3)      # Load a word from memory at byte address labelled data
22         not   x3, x3         # One's complement x3
23         la    x4, data       # Load x4 with address of data label
24         sw    x3, 4(x4)      # Store x3 to memory at byte address in x4 , offset by 4
25
26         ###    END OF TEST CODE   ###
27
28         # Exit test using RISC-V International's riscv-tests pass/fail criteria
29         li    a0, 0          # set a0 (x10) to 0 to indicate a pass code
30         li    a7, 93         # set a7 (x17) to 93 (5dh) to indicate reached the end of the test
31         ebreak
32
33         # ----------------------------------------
34         # Data section Note starts at 0x1000, as
35         # set by DATAADDR variable in rv_asm.bat.
36         # ----------------------------------------
37         .data
38
39 # Label for the beginning of data
40 data:
41         .word 0x12345678     # Test word data value
```

Lines 20, 22, 23, 29 and 30 use pseudo instructions. At lines 20 and 22, with the load address (**la**) instruction, these are two instruction pseudo instructions. The full list of standard RISC-V pseudo-instructions is given in volume 1 of the RISC-V specification (Ch. 25).

Compiling Code

At this point we have enough to construct and compile (assemble) a program. In this and the next section I will be going through the steps to be able to compile code and then run it on the ISS. For those who don't want to take this step just now, you can skip these sections and go straight to the Conclusions section at the end. The instructions here are all based on using the RISC-V tool chain for Windows 10, and the wyvernSemi riscV project's ISS, which has the example code, discussed above, included in the bundled.

Getting the toolchain

The RISC-V gnu toolchain can be downloaded from here, and needs to be installed at a convenient place. The default is `c:\SysGcc\riscv`, though it may be installed elsewhere convenient. If, during installation, the "Add binary directory to %PATH%"

was ticked, then the setup is completed after installation. To check if you have this installed, open a new console and type at the prompt "`where riscv64-unknown-elf-as`" and you should get a response something like:

```
c:\SysGcc\riscv\riscv64-unknown-elf-as.exe
```

If it wasn't found something went wrong.

The ISS bundle can be downloaded from here. Unzip this in a suitable folder. The ISS executable `rv32.exe` uses visual C++ redistributable libraries. If you have these installed then running the ISS with `rv32 -h` will display a help message, otherwise it will give an error message. The redistributable installation packages are bundled with the ISS to install if not done so already. If you have visual studio you may compile from the source directly, which can be found on github.

Compiling

A simple batch file is included with the ISS bundle to compile code, called `rv_asm.bat`. This can be used with a single argument to specify a file to assemble to a RISC-V executable called test.exe. This can be used to compile the provided examples: `example.s` and `example_pseudo.s`. E.g., `rv32_asm.bat example.s`.

I could skip over what this batch file does, but it is worth taking a look at the compilation process. The toolchain has a twostep process in order to compile an executable. There is the assembler program (`riscv64-unknown-elf-as`) and there is a linker (`riscv64-unknown-elf-ld`). The `riscv64-unknown-elf-` prefix is there to uniquely identify the programs for the RISC-V toolchain. Though we have a 64 bit toolchain (`riscv64-`) it can be restricted to 32-bit instructions a we'll see, and it is not specific to a particular RISC-V system or board (hence `unknown-`). The `elf-` indicates that it uses "executable and link format" for the compiled code—a very common format which defines the sections we discussed earlier (data, text, bss etc.), optional symbols to help debug the code as well as the actual program and predefined data. There are versions of the assembler and linker without these prefixes in the toolchain folders, but many developers have multiple toolchains installed and using the prefixes avoid clashes.

The batch file takes care of the prefix, so let's pretend we don't need it and see what the two steps are with all the fluff removed:

```
as -fpic -march=rv32i -aghlms=test.list -o test.o example.s
ld test.o -Ttext 0 -Tdata 1000 -melf32lriscv -o test.exe
```

The `as` executable assembles the source (`example.s`) to an "object file". An object file is an intermediate compilation point, also part of higher level languages, where

the instructions have all been encoded, but any reference to absolute addresses is yet to be determined and added to the code. This means that the code is locatable anywhere in memory, so that the linker can decide where this is when constructing an executable from one or more object files. It is the linker that will provide the final absolute addresses and complete the compilation process. The object files will be the same format whether assembly language was the source code or C or whatever. Thus, the linker is language independent. If an object file created from C++, say, is disassembled (more later) it will show processor instructions just as if it was created from an assembly program.

Looking at the assembler's arguments the first is `-fpic` to specify position-independent-code. We will talk about this shortly, but the example code can be assembled without this argument. When compiling code, especially higher level languages like C, when position-independent, the compiler will only use instructions that are relative to the PC, making the code relocatable within memory. In this tool chain, to aid in doing this, a 'global-offset-table' is used, which keeps all the offsets required within the code in order to reference instructions for jumps and branches and for data. In the assembly language there is even a `.got` section (cf. `.data` and `.text`) where this table data is kept. For our example, this makes little difference.

The `-march=rv32i` argument restricts the assembler to accepting only RV32I base instructions (and *Zicr*). For assembly this means that it will give an error if, say, a multiply instruction from the RV32M specification was used. The compiler for C (or other language) uses this argument to restrict which instructions it will use to map the source code, making it compatible with the target processor's specification. A processor that has RV32M instructions can then have source code compiled, as an example, with `-march=rv32im`.

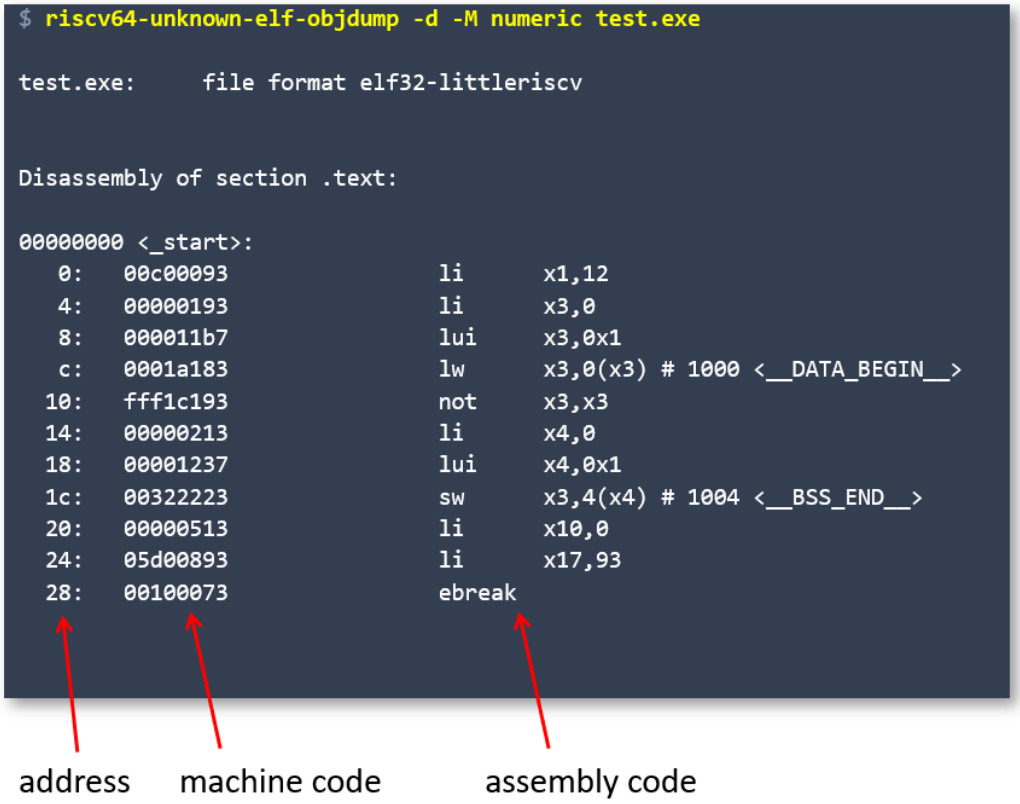
The next argument `-aghlms=test.list` specifies that a list file is to be generated to show information about what was assembled. I won't go into details about what all the letters after the 'a' mean, but the example pretty much turns on everything. When run, a new file `test.list` is created, as specified, which has a listing of the code, now with the *provisional* address locations and machine code and other useful information. The `-o` option specifies that the output will be in a file `test.o`, and finally the source code is specified. Multiple source code arguments could be specified for inclusion in the object file, though more usually source files are compiled to objects separately and combined with the linker (see below).

The linker (`ld`) takes the `test.o` object file and generates the executable. It can take multiple object files as input to create a single output—this is the 'linking' part of the operation where it links together several objects, allocating position in memory for

the code and data and finalizing absolute addresses. In the example, both the text and data start addresses are defined with `-Ttext 0` and `-Tdata 1000`. The next argument, `-melf32lriscv`, tells the linker that the objects are 32-bit RISC-V code (as opposed to, say, 64-bit). Finally, the output file is specified with a `-o` option, in this case `test.exe`. It is this executable that can be loaded to a RISC-V system's memory and executed by the processor.

Disassembling

When the example code was compiled a `test.list` listing file was produced, giving a lot of information prior to linking. It may be that there are objects files and executables to hand that do not have this listing (perhaps they were precompiled). We can still get information about them by disassembling them. The `objdump` (prefix assumed) executable will do just this. The diagram below shows the example program disassembled using this program:

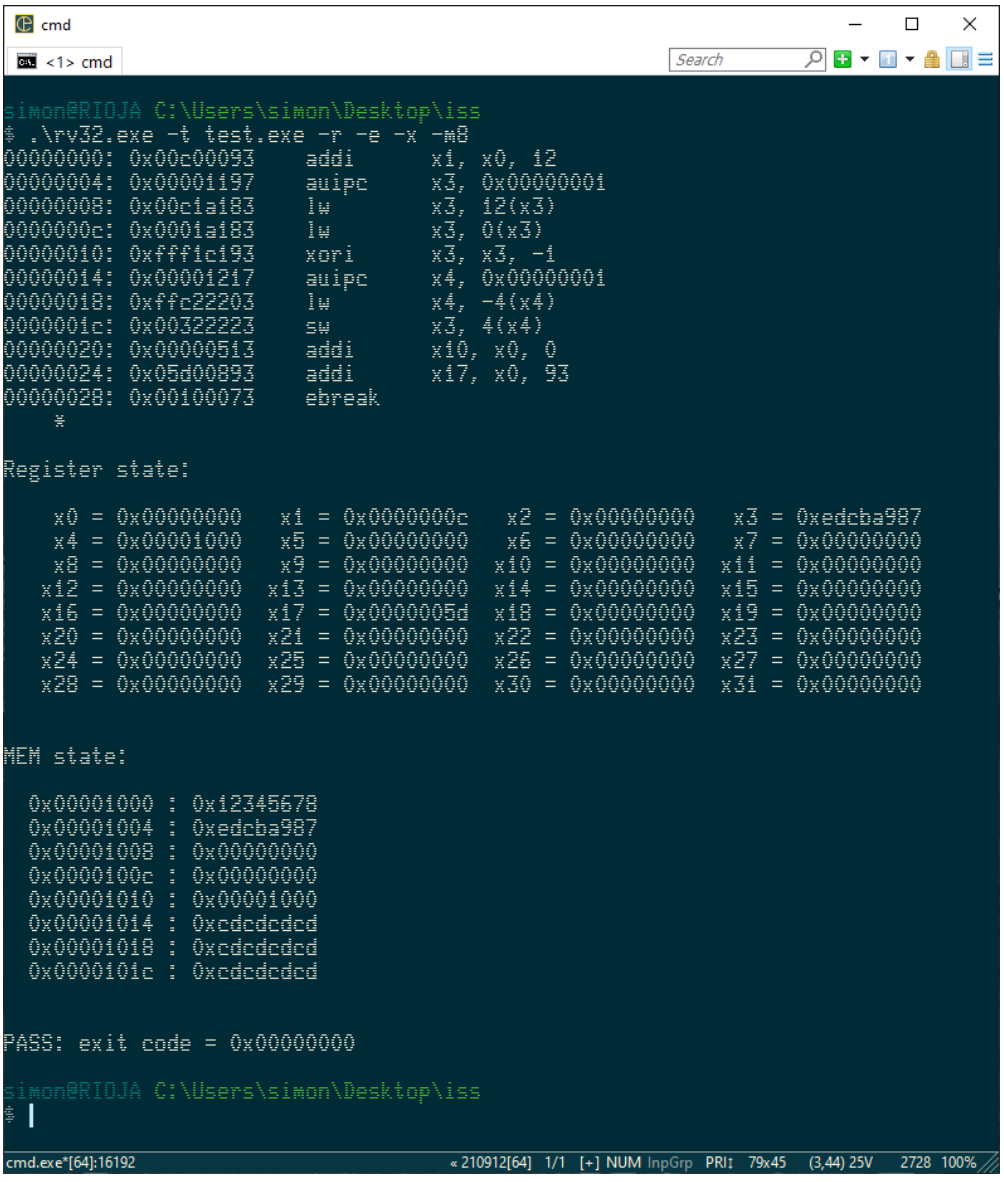


The `-d` option tells the program to disassemble the text section of the file. If a `-D` option was used instead, it will attempt to disassemble all sections, including data—which may or may not be useful. The `-M numeric` option tells the program to display instructions with the registers as the indexed *x* values. Without it, the ABI names are used instead. In this example the `test.exe` executable was disassembled, but the `test.o` object file could just as easily have been disassembled. There are plenty

more options for displaying data from other sections, the global symbols, etc. If you execute the program without any arguments, it will list what all these options are.

Running Code

Having generated an executable, we can try running it on the ISS. The diagram below shows an example of running the `test.exe` executable on the ISS, this time compiled from the `example_pseudo.s` code:



The ISS has various options (use `rv32.exe -h` to list them all) but the ones used here load the ELF executable to memory (`-t test.exe`), enable run-time disassembly (`-r`), specify to halt on **ecall** (`-e`), dump the registers when complete (`-x`) and dump the first 8 words from data memory when complete (`-m8`). If the ABI register names are required, a `-a` option can be specified.

Having run the test program, we can now inspect what is in the registers and what is in memory and see if this matches with expectations. Normally a test program would be self-checking and produce a pass/fail criterion on completion, such as a 0 in *x10* and 93 (`0x5d`) in *x17*. This, in fact, is what the ISS is looking for when it exits to give a pass or fail message. This is copied from the exit pass/fail criteria from the RISC-V International's unit tests, which the ISS has run.

More information on the ISS is given in it's manual, but one last thing to mention is that the ISS has a model of a 16550 UART built within it. The upshot of this is that if one writes a byte to the transmit-holding-register (THR) at address `0x8000000`, the ACSII character will be printed to the screen. This, then, gives a means for the programs themselves to display messages. Why not create some programs, maybe with branches and jumps included, that also display messages to the screen, perhaps by putting predefined strings in the data section.

Conclusions

In this last part of this document on processor design, assembly language programming has been introduced. The RISC-V has, once again, been used as a case-study looking at how the instructions presented in part 2 can be programmed in assembly, along with all the directives for defining text and data sections, as well as macros, constants and fixed words, bytes, and strings.

The main aim has not been to give a definitive guide to RISC-V assembler but to allow enough of a start for a reader to get on with trying some programming for themselves to consolidate all that's been learnt over this series. For those intrepid explorers who want to get straight on and have a go, the RISC-V toolchain was introduced to assemble code, along with an ISS to run it and experiment with new instructions and results.

Assembly language is the first step on the software side of processor design, which is a huge subject in itself. In the future I plan some more documents that bridge the gap between this lower level programming and higher level languages referencing stacks, heaps, and other memory organisation, as well as calling conventions alluded to with the RISC-V ABI. I hope this series has been informative, especially for those new to the subject and that the pace and amount of information has been judged right to allow an individual to start exploring further for themselves.