RISC – V (reduced instruction set computer) as the name implies is made of an Instruction set architecture (as in what instructions the computer can operate) that is very simple and expandable. The base RISC V core only comes with base integer instructions to do arithmetic operations and transfer information between registers. Any other functionality is added from different extensions labelled by letters I, M, A, F, D, C, S and L. I is the base instruction set M is for multiplication and division and A is for atomic functions. The standard core has 32 general purpose registers which can be used to execute the instructions which is given to the processor in memory.

Registers

In the RISC V processor, we have access to 32 general purpose integer registers of with a specified width (32 bits in RV32 and 64bits in RV 64).A register is used to store information temporarily whilst the program is running. So, all the instruction rely around reading from and writing to registers. In the RISC V processor, we have access to general purpose integer registers which are used for the executing the instructions written in the program. They store the inputs and answers to calculations done by the processor. Each register has a different typical role The first one x0 is wired to 0(therefore can only be read from) , 7 temporary registers, 8 for function arguments, 12 to store information, a stack pointer, global pointer and thread pointer and a program counter. The temporary registers are used with functions(subroutines) for local variables that are not needed in the main program. The stack pointer is used to store the base address of a stack stored in the memory, which can be used to return to previous states after a function has finished. Global variables are stored in memory, so the global pointer stores the address at which these variables are located. A thread pointer refers to variables stored in memory that are specific to its operation. The table highlights how the physical registers are split for these functions. XLEN is the register width

Register Name	ABI Name	Description
x0	zero	Hard-Wired Zero
x1	ra	Return Address
x2	sp	Stack Pointer
x3	gp	Global Pointer
x4	tp	Thread Pointer
x5	t0	Temporary/Alternate Link Register
x6-7	t1-t2	Temporary Register
x8	s0/fp	Saved Register (Frame Pointer)
x9	s1	Saved Register
x10-11	a0-a1	Function Argument/Return Value Registers
x12-17	a2-a7	Function Argument Registers
x18-27	s2-s11	Saved Registers
x28-31	t3-t6	Temporary Registers

Table 1.1: RISC-V Base Integer Registers Of Size XLEN

CSR register

Register	Description
misa	Machine ISA
mvendorid	Machine Vendor ID
marchid	Machine Architecture ID
mimpid	Machine Implementation ID
mstatus	Machine Status
mcause	Machine trap cause
mtvec	Trap vector base address

Register	Description
mhartid	Machine Hardware thread ID
mepc	Machine exception program counter
mie	Machine interrupt enable
mip	Machine interrupt pending
mtval	Machine trap value
mscratch	Scratch register

Table 1.3: RISC-V Machine Mode Registers

The calculations are operated by the 32 general purpose registers but to observe and interact with the state of a processor we interface with Control and Status Registers (CSR). There many such CSR registers tell us information about the current state of the processor. There are registers which only provide information about the chip and others that convey current state of the processor as it tries to execute code. E.G. MISA only provides information as it only conveys what the register width is and what extension have been enabled. Other registers such MSTATUS change to reflect the state of the processor as it tries to execute code. We can look at the values stored in CSR's to check if an error (exception) had occurred and additional information can be gathered in MCAUSE or MTVAL to find where and what caused the exception .Some CSR's contain information that needs to be updated often (E.G MSTATUS) or not updated at all (E.G MISA) but most only contain certain bits that change so inside each CSR it is split into fields which signify how the bits in the register are organised for the encoded message. Each of fields operate in different modes; WIRI (register does not respond to reading or writing), WPRI (we can write to the register but we need to ensure this section is unaltered), WLRL (only accepts certain legal values and erroneous output otherwise when read) and WARL (where we can write anything but if the value that is not legal then the output is the previously stored value), when reading from or resetting such CSR's we must check what the type of fields it has to ensure that we don't raise any exceptions by trying to write the wrong values. Some example as to what type of data is stored in CSR's is the **privilege level** of the program ran. The privilege level of a program indicated what level of access the program has to the processor's resources. In RISC we have 3 such levels user, supervisor and machine with user being the most restricted and machine mode being completely trusted. The privilege level is stored inside MSTATUS. Initially without any modification all software run on machine mode.

CSR commands

If we wanted to change or read the information in CSR's we need to use CSR instructions, with these instructions we can read and write to CSR's. There are 2 further types register-register instructions (which only deal with registers and uses the values stored within them) and Immediate instructions (which directly involve the number in the code itself).

Register-register instructions

- csrr destinationregister csr – this instruction reads the value in the csr register and stores it in the destination register usually the general-purpose registers
- csrrc destinationregister csr maskregister – this copies the value of the csr into a register, using another integer register as a mask, turns wherever the mask is 1, turns the same bitposition to 0, clearing that bit.
- CSRRW rd csr rs2 – copies the valuel in csr to rd then writes the value of rs2 to csr.
- CSRRS rd csr rs2 – copes the value in csr to rd, then using rs2 as a mask turns wherever rs2 has a 1, in the corresponding bit in the csr is also turned into 1.

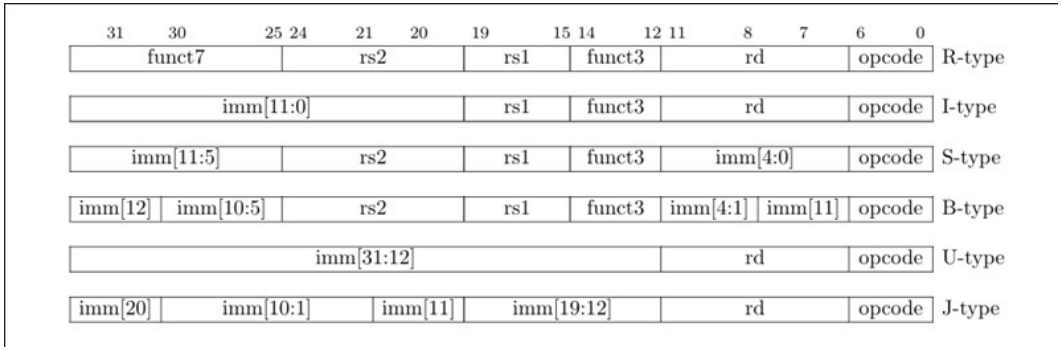
Immediate instructions

- Csrrci rd csr imm – the same crrsc but uses a number instead of reading it from rs2.
- Csrrsi rd csr imm – same as crrs but with a number
- Csrrwi – same as csrr.

Imm has the limitation that it can only have a maximum of 5 bits in the specific instruction format, so the maximum value of imm is 31 but the register width is more than 5 bits and in that case the answer is calculated and zero-extended. As in 0's are inserted after the number to make up the number of bits needed to store the number.

Instruction formats

Instructions in the ISA follow certain formats which dictate how they are written binary and what each bit represents, although we don't have to write in binary it does inform us the limitations of instructions such as the range of values for immediate.



There are 6 different types of instructions in the ISA, with each instruction being 32 bits long. Here the imm refers to the intermediate which is a number. rs1 and rs2 are source registers used in the operation. rd is where the result is stored. Opcode and funct refers to identifiers used by the processor to find out what instruction to follow, although this information is given it is not necessary to write all of it when trying to use an instruction as there are so few instructions that keywords itself are enough. The I-type generally refers to the integer instructions. R is register operations, S is for store operations, B is for branch operations which determine the flow of code. J is used for jumping between sections of code To invoke functions and use loops. U type is used in instructions such LUI and AUIPC where the bits from a register need to be shifted a certain amount.

Loading and store

Loading is the action of moving data from memory into a register, and storing is moving data from the registers into memory. Information is stored in words with each being 32 bits long or 4 bytes in the RV32. when dealing with commands in Load and Store we **use prefixes of d (double word – 64 bit sequences), w (word 32 bits), h (half word 16 bits) and b (one byte 8 bits).** When trying to load or store data we need to ensure that we provide the exact space needed to store the data and that the address of memory is aligned with the data size, as in the memory allocated to this data is of a size that is a multiple of the data size being stored. We also have versions of the instructions for signed and unsigned handling of the numbers. Denoted by the letter U.

Instruction format for loading (use L) for storing (use S)

- *Function destination/source_register offset(address_of_memory_to_access/write to_from) – the offset works as adding that number onto the address stored and getting the data stored there.*
- We also have LUI rd imm– this adds the value of imm onto the 20 most significant bits of rd and causing the least 12 bits in a 32 bit register to all be cleared to 0.
- Auipc rd imm – adds the imm to PC(program counter) then stores this answer in the rd. this can be used to address other lines of code based on PC – relative addressing

The ISA also includes what are known as pseudo codes which are like built-in functions in higher level languages. Some in Load and store are;

- MV rd rs1 – this copies the contents of rs1 onto rd
- Li rd imm – writes the value of imm onto the destination register
- La rd symbol – writes the address of a certain symbol (a symbol is an entity in the program like a variable or function)

Usually when loading we fill in the least to most significant bits, except in the case of the LUI where the number is loaded to the highest 20 bits.

Integer instructions

Mostly use the I or R type format, when dealing with immediate it uses the I format and when dealing with registers is uses the r type format, and has the basic functions to

Add, subtract, check if less than, shift bits left or right, and perform bitwise AND, OR and XOR.

They have a general form of:

Function rd rs1(actionon) rs2/imm (the thing the action does) so if had less than then it would mean (rs1 < rs2).

Functions are:

ADD[I], SLT[I][U] , AND[I}, OR [I], XOR[I], SLL[i], SRL[i], SRA[i], SUB. – *I meaning immediate and U for unsigned*

- SLT stands for Set if less than. – this checks if the first source register is less than the intermediate or the second source register
- SLL - Logical Left Shift – this shifts the value in the first source register, a certain number of times based on the number in the second source register or immediate
- SRL – Logical Right Shift – same as SLL
- SRA – Arithmetic right shift – similar to SLL and SRL but retains the sign of the original value.
- When shifting the bits the empty bit positions are set to 0 in SLL and SRL.

If an overflow occurs when adding numbers, the lower bits are stored until the register is full.

Control transfer instructions

Control transfer instructions are used to control the order of code executed and is used to jump to functions(subroutines) or in loops

These follow the B and J formats; B format is used in the conditional transfer and j type is used in the unconditional format. In the B format we compare 2 registers and if the result is true, then we jump where the address points in the code. We there many conditions but the basic ones are;

BEQ – is equal to

BNE – is not equal to

BLT – is less than

BGE – is greater than or equal to

When using unconditional jumps we have;

JAL rd offset(address_to_jump_to) This records the address of the next line of code, then jumps to the address specified in the offset

JALR rd rs1 offset(address_to_jump_to) this records the address of the next line of code, then jumps to the address based on the offset + rs1.

Once we have executed the function block, we can say *RET* to return to the address stored in rd.

If you don't want to return back, we can use the pseudo-operator *j* address to only jump the specified line.

Multiplication extension

this adds the instructions to multiply and divide numbers using the R type format with several functions;

MUL – multiplies the 2 numbers resulting in a 2x bitlength so the lower bits are stored

MULH – stores the higher of the 2x bits

MULHSU – stores the higher of the 2 bits but using unsigned numbers, the S indicates a signed character, it is necessary as here we are multiplying a signed and unsigned value

MULW – the W is for multiplying 2 32-bit numbers in a 64bit registers so the full answer is stored

DIV – divides and stores the integer answer

DIVU – divides unsigned

REM – gives the remainder

REMU – same but unsigned

When try to divide by 0, instead of giving an exception we get an output;

Condition	Dividend	Divisor	DIVU	REMU	DIV	REM
Division by zero	x	0	$2^{XLEN}-1$	x	-1	x
Overflow (signed only)	-2^{XLEN-1}	-1	–	–	-2^{XLEN-1}	0

atomic memory operations

Part of the A extension of the ISA, these instructions are used to read and modify the data in memory, they follow the R format, and generally take the data from memory into a register, perform the activity and store the result in another register then write that to the memory in the same address. An operation is atomic if the operation can occur with any interruption examples of such operations are the LOAD and STORE functions where data is read or written to memory in one step. These operations are very useful to ensure proper memory space is allocated. An example would be to store the answer to an operation in the same memory location as from where we read the input. To ensure we can do this consistently we need to lock the memory address during this operation and then release it, using the prefixes .aq(acquire) and .rl(release). as part of the A extension.

AMOSWAPP – swaps the value in memory with another value stored in a another register

AMOADD – adds the number from the other register onto the value in memory

AMOAND – performs bitwise AND on the value in the memory and register

AMOOR – performs bitwise OR on the value in the memory and register

AMOXOR – performs bitwise XOR on the value in the memory and register

AMOMAX [U] – writes the larger value between the memory and register

AMOMIN [U] – writes the lower value between the memory and register