1 Week 1/2

1.1 Introduction to RISC-V

1.1.1 Binary Representation

All information passed through a processor can be expressed as a sequence of binary digits.

Conversion between a base 2 number and a base 10 number can be done rather efficiently:

$$1001\,0111_2 = 151_{10} \tag{1}$$

	1	$\times 1$	1
	1	$\times 2$	2
	1	$\times 4$	4
	0	$\times 8$	0
	1	$\times 16$	16
	0	$\times 32$	0
	0	$\times 64$	0
+	1	$\times 128$	128
			151

1.1.2 Hexadecimal Representation

We must also recall the hexadecimal representation of numbers that makes binary just a bit easier to look at.

1.1.3 Two's Complement

This is a convention for the expression of signed numbers, such that the most significant bit is negative instead of positive.

$$1001\,0111 = -105_{10} \tag{2}$$

	1	$\times 1$	1
	1	$\times 2$	2
	1	$\times 4$	4
	0	$\times 8$	0
	1	$\times 16$	16
	0	$\times 32$	0
	0	$\times 64$	0
+	1	$\times$**-128**	**-128**
			-105

1.2 Data in Memory

All elements in a computer's memory is labeled with a memory address and the units of data in a computer's memory is a byte, which is 8-bits and is represented (usually) as two hex characters $\texttt{0xFF}_{16} = 1111\,1111_2 = 255_{10}$.

Each memory address is represented by a 4-byte word (32-bits) which are arranged sequentially for the given process

Note 1 (Word) *It should be remembered that the 32-bit piece of memory that we have is called a 'word' in RISC-V 32 IF (which is the complete name of the instruction set we are dealing with). This is easy to remember since the processor acts on 32-bit instructions. More on this later.*

Note 2 (Memory Allocation) *Remember that the allocation of this memory is handled by the operating system, this is outside the scope of this class.*

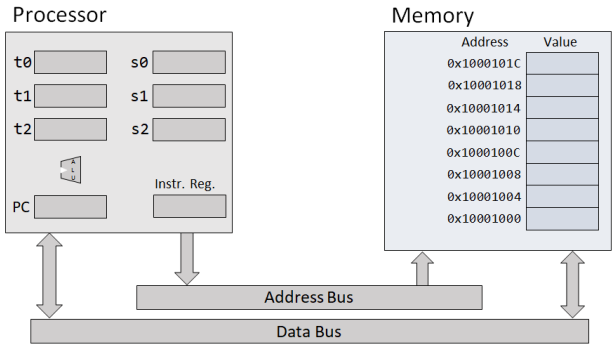

Figure 1: The organization of the processory and memory in a conventional computer detailing the contents of the processor registers and memory address-value map

1.3 Registers

Registers are given a name in assembly, we will come back to this later and by now you already know about them.

They are just a whole bunch of faster-to-access storage locations for data and hold 1 word worth of data.

1.3.1 Endian-ness

RISC-V is little-endian, which means that we read the contents of memory from right to left. This is nice for the processor but not nice for us.

What little endian means is like this:

$$0x \quad 12 \quad 34 \quad 56 \quad 78$$
$$\uparrow$$

The byte with the arrow pointing to it is the first byte in the sequence, we then read to the left by **byte** (not bit). So, for example, if we stored the word 'BEAN' into our 32-bit word, we would see this:

$$0x \quad N \quad A \quad E \quad B$$
$$\uparrow$$

If we were to represent the word in ASCII.

Note 3 (ASCII Characters) *The ASCII alphabet contains 256 characters, thus each ASCII character is represented in 1 byte (8 bits).* ***Excersize****, what is the binary representation for the 256th ASCII letter.*

If we take a look at the RISC-V Emulator, RARS, we can get a feel for a larger example of the little endian nature of riscv.

1.4 Computer Organization

1.4.1 The Processor

The processor is the brain of the computer and is used to perform the computations that we give it, we will learn more about the performance of the processor in the 3.1 section. Some important things to note about the processor[1]:

- **Registers**: The fastest possible data access location.
- **I/D Cache**: The instruction and Data cache, they store information that is essentially "currently in use" (NEED VERIFICATION).
- **L0 cache**: Note that some computers do not have an L0 cache. This would be a cache to which the operational units have direct access. On the order of 128 bits wide.

[1]These are mostly supplementary and begin diving into the nitty-gritty of how a processor is architected. I grabbed most of my information from here.

- **L1 cache**: Typically the "on-chip split instruction and data caches" or "unified on-chip cache". These are fast caches that run at the chip clock speed and can be accessed within one cycle. Between 8kb and 32kb (lower case b is 'bits').
- **L2 cache**: An external cache that is much larger (256kb to 2Mb) that can be accessed within some multiple of the CPU clock speed.
- **L3+ cache**: There can be many levels of cache, they generally get progressively slower and larger.

There are also some special registers in the processor that do important things:

- **Instruction Register**: Keeps track of where we are in the program, more about this later. In RISC-V this is the program counter `pc` register.
- **Global Pointer**:
- **Return Address**: The address that stores the address to return to after an internal function call. More about this later.
- **Stack Pointer**: The pointer to the base of the stack at the current moment. More about this later too. `sp`
- **Frame Pointer**: The pointer that points to the base of the current frame `s0, fp`. This can double as another saved register in optimized code and is only ever used for debugging purposes.
- **Thread Pointer**: `tp`
- **Temporary Registers**: `t1 - t6`, for storing temporary data that can be erased at any point. Volatile registers.
- **Saved Registers**: `s1 - s11`, for data that need to be preserved across function calls and exceptions.

1.4.2 Memory

This is essentially just a giant map of data with key-value pairs that represent certain data.

1.4.3 Data Bus and Control Bus

1.4.4 Storing and Loading from memory

The two basic loading and storing operations in RISC-V are the `lw,␣sw` instructions which mean load-word and store-word respectively.

2 Week 3

2.1 Instruction types and Immediates

2.1.1 Representing Instructions in Binary

In order for a processor to understand what we want it to do, we need to convert our instructions into a binary format. Assembly language is one level abstracted from this binary representation and it is the job of the assembler to translate our code into a machine readable binary representation. Luckily for us, this translation is almost direct, so we can get to understand what is going on under the hood of the assembler.

There are some standard parts to the instructions that we need to recognize to fully understand what is happening:[2]

- **opcode (7)**: The opcode gives the processor the first indication of what instruction we want it to perform.

[2]This can be found on page 89 of (The Morgan Kaufmann Series in Computer Architecture and Design) David A. Patterson, John L. Hennessy - Computer Organization and Design RISC-V Edition, our class textbook.

- **func3 (3), func7 (7)**: The func3 and 7 parts along with the opcode specify exactly which instruction we are asking the process to do. The reason that these are placed apart (as you will see in the diagrams below) is because some types of instructions don't need the full func7 specifier and the splitting of these parts simplifies the development of the hardware. Additionally, extensions of the ISA support 16-bit instructions so the RISC-V architecture by default reads 16-bit chunks of data.
- **rd (5)**: The destination register, this is where the result of the register is stored if necessary.[3]
- **rs1, rs2 (5)**: These are the argument registers. rs1 is the first one we type and rs 2 is the second, so in the command add t1, t2, t3, t1 is r1 and so forth.5

[3]You can really dive into this seperation thing, there si a presentation here and a very good stack exchange answer here and a decent little debate with an insightful comment here

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct7							rs2					rs1					funct3			rd					opcode							

Figure 2: R-Type instruction layout

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s*	imm[11:0]											rs1					funct3			rd					opcode							

Figure 3: I-Type instruction layout, * the sign of the immediate, if loaded into a memory location, the immediate is sign extended down to bit 11 as shown in figure fig. 4

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sign																					imm[11:0]											

Figure 4: Representation of an 11-bit immediate in memory. Unsigned immediates are represented in the same way, simply adding one bit to the significant end.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s*	imm[11:5]						rs2					rs1					funct3			imm[4:0]					opcode							

Figure 5: Representation of the S-Type instructions, note that the immediate is still 11 bits, it is simply split over two parts. Similarly to the I-type instructions, the immediate representation will follow the outlined form above.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s*	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	rs2							

Figure 6: Representation of the S-Type instructions, note that the immediate in this case is 12 bits, and is again split over two parts. The reasoning behind this ordering can be found in chapter 4.4 of the textbook.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[31:12]																				rd					opcode							

Figure 7: Representation of a U-Type instruction in binary.

Name	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-Type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic Instruction format
I-Type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-Type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	Stores
SB-Type	imm[12, 10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode	Conditional Branch Statements
UJ-Type	imm[20,10:1,11,19:12]				rd	opcode	unconditional jump format
U-Type	imm[31:12]				rd	opcode	Upper immediate format

2.1.2 R-Type Instructions

These are operations that work on multiple registers without any intermediates or interaction with memory, such as `add,sub`.

2.1.3 I-Type Instructions

2.1.4 S-Type Instructions

2.1.5 B-Type Instructions

2.1.6 U-Type Instructions

2.1.7 JAL-Type Instructions

2.2 Functions in RISC-V

Recall the calling conventions.

3 Week 4

3.1 Performance

Key formulae:

$$\text{Performance}_X = \frac{1}{\text{Execution Time}} \tag{3}$$

$$\text{CPI} = \frac{\text{Clock periods}}{\text{Instructions}} \tag{4}$$

$$\text{CPU Time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \tag{5}$$

4 Week 5

4.1 Pointers

4.2 Computer Arithmetic

5 Week 6

5.1 Compiler Structure

5.2 Exceptions and Interrupts

5.3 I/O Devices

6 RISC-V Green Sheet (but better)

MNEMONIC	FMT	NAME	DESCRIPTION	NOTE	TYPE
add, addi					
and, andi					