

Hackathon  
→ Custom Acceleration of Large Language Model Primitives

Studio Version 9.4.2

Release Date March 2025  
  
Author: Tadej Murovič, [tadej.murovic@codasip.com](mailto:tadej.murovic@codasip.com)

## 

# **TUTORIAL and INSTRUCTIONS**

1. **ACCESS**

Each participant will need to provide their name and email address. With this information, we will create a Codasip Cloud Studio account.

The instructions are:

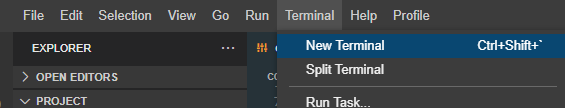
1. The temporary password, which needs to be changed, is: 0000
2. Studio can be accessed through:<https://cloudstudio.studioin.cloud/> or<https://studioin.cloud/>, with the username to be decided.
3. **TUTORIAL**

<https://drive.google.com/drive/folders/1Dn0m6Tm7fSS0ezPueAvDWqsJhdhM-70I?usp=sharing>

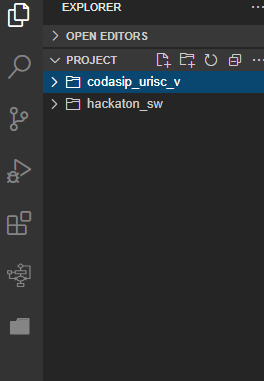
This tutorial (courtesy of Keith Graham) will take you through the basic principles of using a studio in the cloud. This should preferably be done before the hackathon starts.

1. **SETUP**

Afterwards, go to <https://github.com/TadejMurovic/riscv_academy_hackathon_april_2025> and run the commands in the terminal of Studio in the cloud. If no terminal is seen, go to terminal→new terminal.



In the end, the project explorer should look like this:



1. **HACKATHON STARTUP**

Codasip Studio is a tool used to develop CPUs and software. There are two visible projects: hackathon\_sw (SW) and codasip\_urisc\_v (HW). In the hackathon\_sw project, you will find the testing suite and the annotated part of the code that you need to accelerate to the best of your abilities. In the codasip\_urisc\_v project, you will find the CPU description and model which will run your code. Studio enables SW/HW co-design, meaning you can modify the behavior of the hardware to best suit your algorithms. To keep things simple, you will develop custom single-cycle instructions.

As you saw in the previous tutorial there are many things to do in Studio, but for time’s sake we will only focus on a couple of functionalities.

1. Under the explorer tab you can browse your project files. 
2. You can check the initial transformer implementation under hackaton\_sw/src/main.c
3. The CPU custom instruction file is under codasip\_urisc\_v/model/share/isa/isa\_hackaton.codal

***Software***

In our hackathon c-code, we have implemented a dumbed down and dummy transformer encoder. The simplifications are there to enable easier and faster iterations. These changes do not detract from the gained experience, as identical mathematical operations are present in real life LLMs (only with many more and bigger dimensions). The high level operations are:

1. **Input**: 3 tokens, each is a 4-dimensional vector.
2. **Compute** Q, K, V with WQ, WK, WV (all 4×4 matrices).
3. **Self-Attention** (single head only, for simplicity).
4. **Residual connection**.
5. **Feed-Forward** sublayer (2-layer MLP).
6. **Another Residual.**
7. **Output final 4D embedding for each token**.

We do not show training/backprop or multi-head attention.

### **Hyperparameters**

#define SEQ\_LEN 3 // We have 3 tokens in our sequence

#define MODEL\_DIM 4 // Each token is a 4D embedding

#define FF\_DIM 8 // Hidden dimension for feed-forward

In a real Transformer, MODEL\_DIM could be hundreds or thousands (e.g. 512, 1024). Here we do 4 for simplicity.

### **The Input Tokens**

int16\_t input[SEQ\_LEN][MODEL\_DIM] = {

{0.1f, 0.2f, 0.3f, 0.4f},

{0.2f, 0.1f, 0.5f, 0.3f},

{0.5f, 0.9f, 0.1f, 0.0f}

};

Imagine these are embeddings for words (or subwords). Each row is a token’s 4D embedding.

### **Q, K, V Weight Matrices**

int16\_t WQ[16] = { ... }; // 4x4

int16\_t WK[16] = { ... }; // 4x4

int16\_t WV[16] = { ... }; // 4x4

These transform each token’s embedding into a Query, Key, and Value.

### **Compute Q, K, V**

int16\_t Q[SEQ\_LEN][MODEL\_DIM];

int16\_t K[SEQ\_LEN][MODEL\_DIM];

int16\_t V[SEQ\_LEN][MODEL\_DIM];

for(int i = 0; i < SEQ\_LEN; i++){

matvec\_mul(WQ, input[i], Q[i], MODEL\_DIM, MODEL\_DIM);

matvec\_mul(WK, input[i], K[i], MODEL\_DIM, MODEL\_DIM);

matvec\_mul(WV, input[i], V[i], MODEL\_DIM, MODEL\_DIM);

}

For each token i, multiply by WQ (size 4×4) to get Q[i], and similarly for K[i], V[i].

### **Single-Head Self-Attention**

We have a function single\_head\_attention(Q, K, V, attn\_out) which does:

1. Dot Product Scores: For each token i, compute score[i,j] = Q[i] dot K[j].
2. Scale: Multiply by 1 / sqrt(MODEL\_DIM) = 1/2 in our case.
3. Softmax: Convert scores to probabilities.
4. Weighted Sum of V: For token i, gather a weighted sum of all V vectors (the weights come from the softmax scores).

In code:

for(int i = 0; i < SEQ\_LEN; i++){

// compute scores vs each j

for(int j = 0; j < SEQ\_LEN; j++){

scores[j] = dot\_product(Q[i], K[j], MODEL\_DIM);

}

// scale, softmax, ...

// then attn\_out[i] = sum\_j( scores[j] \* V[j] ).

}

This means token i is looking at all tokens j, deciding how much each one matters.

### **Residual Connection (After Attention)**

for(int i = 0; i < SEQ\_LEN; i++){

for(int d = 0; d < MODEL\_DIM; d++){

post\_attn[i][d] = input[i][d] + attn\_out[i][d];

}

}

We simply add the attention output to the original input. Real Transformers also do LayerNorm here, but we skip it.

### **Feed-Forward Sub-Layer**

We do this per token:

feed\_forward(post\_attn, ff\_out, W1, b1, W2, b2);

Inside we multiply post\_attn[i] by W1 (size 4×8), apply ReLU, then multiply by W2 (size 8×4), add biases, etc. This transforms each token embedding individually.

### **Second Residual**

for(int i = 0; i < SEQ\_LEN; i++){

for(int d = 0; d < MODEL\_DIM; d++){

final\_out[i][d] = post\_attn[i][d] + ff\_out[i][d];

}

}

We add the feed-forward output to the input of the feed-forward (post\_attn). Again, real code also does LayerNorm.

### **Output**

At the end, final\_out is the updated token embeddings—each now aware of the entire sequence context (thanks to attention).

# **Why Residual Connections?**

Residual connections (also called “skip connections”) are used to:

1. Help with training: They mitigate the vanishing/exploding gradients problem in deeper networks.
2. Preserve original information: Even if the attention or feed-forward transformations drastically alter the embeddings, we don’t lose that initial embedding entirely.

# **Real Transformers vs. Our Dummy Example**

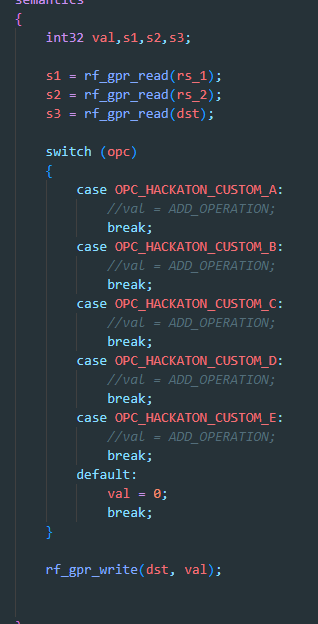
1. Multi-Head Attention: Real Transformers split Q, K, V into multiple heads (like 8 heads). Each head attends separately, and the results are combined.
2. Layer Normalization: After attention and feed-forward, real Transformers do LayerNorm to stabilize training.
3. Positional Encoding: Real Transformers add positional information to the embeddings so the model knows which token is first, second, etc.
4. Training: The weights WQ, WK, WV, W1, W2, etc. are learned via backprop on massive datasets. In our code, we just pick some fixed values.
5. Larger Dimensions: Real models have bigger MODEL\_DIM (like 512 or 1024) and can handle large SEQ\_LEN.

Our code is purely for demonstration of how the data flows in one Transformer block without the complexities of a real production model.

**Your job is to rewrite the functions between the comments “*// START OF HACKATHON CODE*” and”*// START OF HACKATHON CODE*”’. These functions can be rewritten as is, broken into more functions, merged etc.**

**Rules: Redoing the validation calls in main() is not allowed. No direct mapping of inputs to outputs through LUTs is allowed.**

***Hardware***

For the HW part we will focus on adding new instructions to the CPU. The baseline CPU is a [5-stage in-order](http://home.ustc.edu.cn/~louwenqi/reference_books_tools/Computer%20Organization%20and%20Design%20RISC-V%20edition.pdf) RV32I Risc V processor, meaning it is using the base integer instruction set WITHOUT any extensions. ([Link](https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf))  
  
We won’t be adding any standardized extensions, rather you have a template for 5 new completely custom instructions. These instructions use as input registers s1, s2, s3 and write to register s3. For example, you can add:  
  
result = (s1 + s2)/s3;

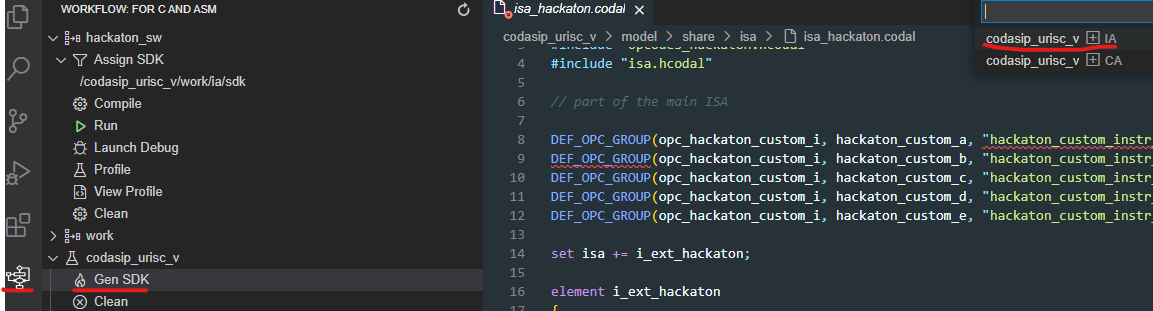
under one of the cases which would create an add and divide instruction.

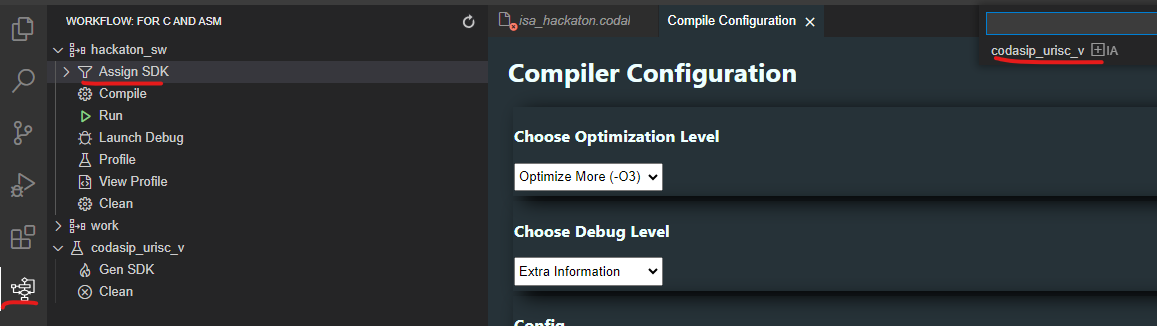
**Your job is to discover which instructions will best accelerate the SW.**

**These instructions will then be automatically used by the compiler and your code should run faster! If the instructions are not detected you’ll have to use inline assembly or simplify the instruction operation.**

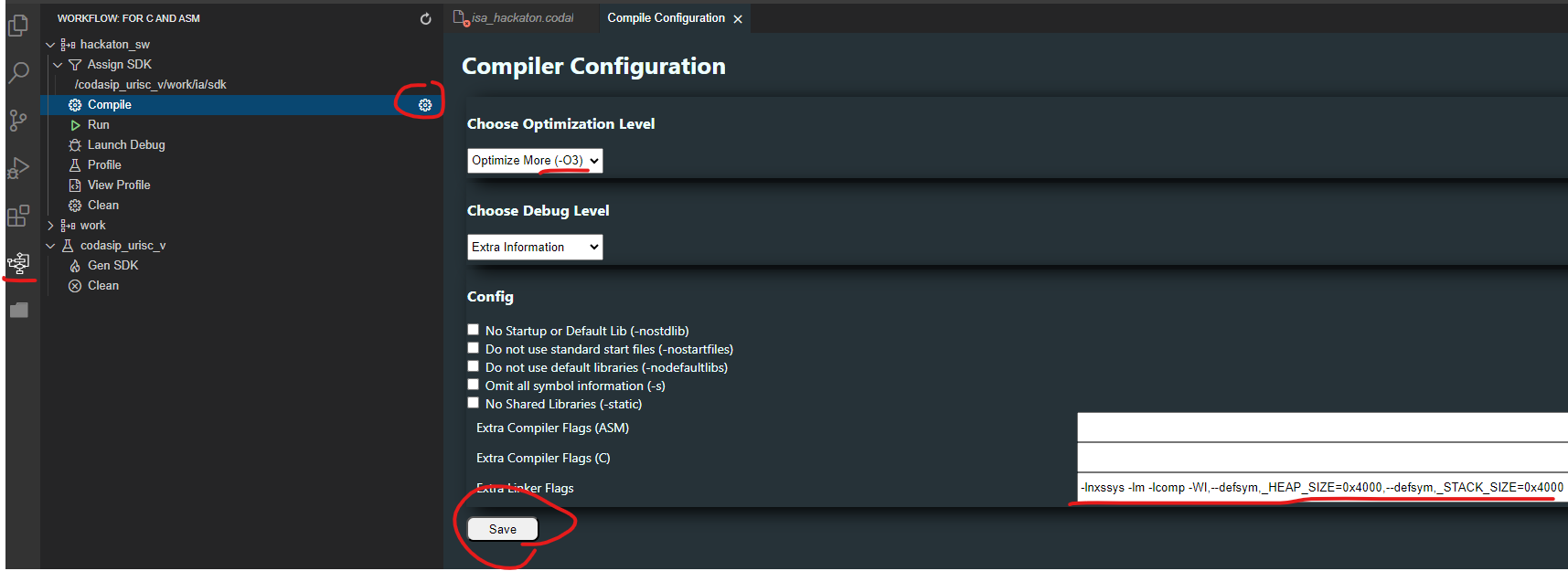
**Rules: Do not touch any other HW components/files as there is not enough time to play around with and debug the architecture of the CPU beyond the execution unit.**

1. **WORKFLOW**

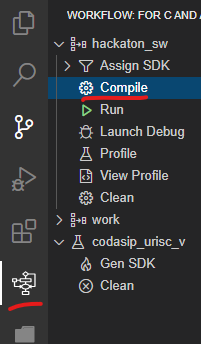
Your work should go as follows:  
  
– A) Build the IA Simulator of the CPU and its CLANG compiler. We won't bother with the CA simulator in our case.  


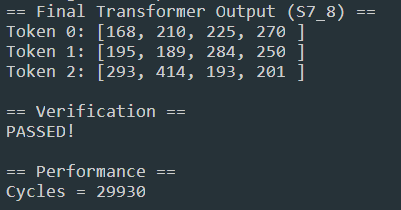
– B) Set the SDK.  


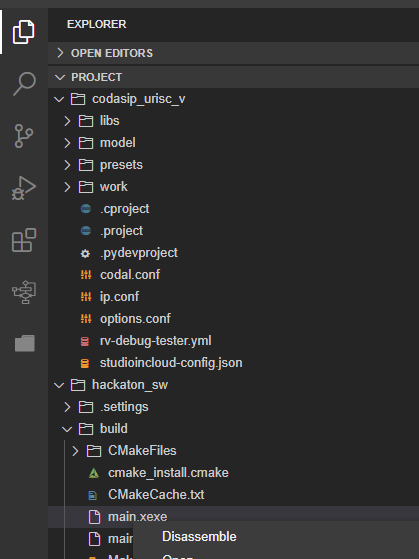
– C) Set linker flags (Default flags to add are: -lnxssys -lm -lcomp -WI,--defsym,\_HEAP\_SIZE=0x400000,--defsym,\_STACK\_SIZE=0x400000). Set optimization to -O3.



– D) Compile your SW with the new compiler.

,

– E) Run inference!  
  
What you want to see is that the tests have PASSED and that the cycle count is as small as possible!

– F) Check the disassembly (right click on .xexe).  


– G) Think about what could be improved! Add/Redo your custom HW instructions and optimize your SW and go back to step A. Iterate! Before recompiling/rebuilding, use ‘clean’ to clean the folders and refresh the site!

1. **RANDOM TIPS**
2. Inline assembly is harder than you think. If you are using the destination register both as an input and as an output accumulator ‘+’ prefix must be used:   
   ***asm volatile ("accumulate %0, %1, %2\t": "+r"(acc) : "r"(x), "r"(y));***
3. The code is using s7\_8 [fixed point](https://en.wikipedia.org/wiki/Fixed-point_arithmetic) variables. It means that 1-bit is the sign bit, 7 bits are the integer part, and 8 bits are the fractional part. Multiplying two fixed point numbers means that the parts get added. So, s7\_8 x s7\_8 = s14.16. This is why we need to shift down (get back to .8) and saturate (keep it inside 7 integer bits).
4. Google SIMD.
5. The compiler is CLANG. Any pragmas you will use need to be clang’s.
6. We are only looking at cycle counts. You can make your HW and Code Size BIG.
7. If your program crashes you’ve done something silly with pointers.
8. You are using the older version of Studio. The frontend is a bit buggy, so be liberal with refreshing everything.
9. Remember to clean both the SW and SDK often to avoid any conflicting generated files.
10. To debug CodAL code use codal prints. For example, codasip\_print(13, “bla bla %d”,var);. Then you need to run the program in the terminal with –logging 13 to get the outputs. For example: codasip\_urisc\_v/work/ia/sdk/bin/codasip\_urisc\_v-ia-isimulator hackaton\_sw/build/main.xexe --logging 13