



Processor Architecture Experiment

מעבדה במעבדי מחשבים - 045117

חוברת ניסוי - מפגש מס' 1

גרסה 1.0

דצמבר 2024

הערות לחוברת נא לשלוח:

לאינה ריבקין – inna@technion.ac.il



- The lab is based on the package being developed by Imagination Technologies and its academic and industry partners.
- The RVfpga system is built around the Chips Alliance SweRVolf SoC, which is based on Western Digital's RISC-V SweRV EH1 core.
- Expected Prior Knowledge
 - Fundamental understanding of digital design
 - High-level programming (preferably C)
 - Instruction set architecture and assembly programming
 - Processor microarchitecture, memory systems (this material is covered in Digital Design and Computer Architecture: RISC-V Edition, Harris & Harris, © Elsevier)



Table of Contents

1.	Historical background	3
1.1.	ISA	3
1.2.	RISC-V	4
1.3.	Integer ISA RV32I	6
2.	Processor Architecture	7
2.1.	Single-Cycle Processor	7
2.2.	Multicycle Processor	7
2.3.	Pipelined Processor	7
2.4.	Superscalar Processor	8
2.5.	Hazards – Data & Control	8
3.	SweRV EH1	10
4.	RVFPGA SYSTEM OVERVIEW	12
5.	RISC-V Assembly Instructions and code examples	17
5.1.	RISC-V Assembly Instructions	17
5.2.	RISC-V main directives	18
5.3.	RISC-V Code Examples	20
6.	Implementation environment for the experiment	22
6.1.	Digilent's Nexys A7 FPGA board	22
6.2.	Development tools	23
7.	Meeting #1 – preparation report	24
8.	Meeting #1 - performing the experiment in the lab	25
8.1.	Programming the FPGA with RISC-V softcore	25
8.2.	SEGGER Embedded Studio – compilation & debugging	28
8.3.	General-Purpose I/O (GPIO)	33
8.4.	Simulation of SW	35
8.5.	C programming	37
8.6.	Hello world from Swerv C program	39

1. Historical background

Development of computers and their processing element started towards the end of WW2. Those machines were mostly special purpose machines working with fixed hard to change programs. However, these machines had all components which make up a computer,

- a central processing element CPU (ALU and Control)
- some program storage
- some way of storing data
- input / output units

The first breakthrough in the development of more universal computers came as 'von Neumann' and his colleague introduced three changes:

- having a set of defined operations (instructions) to work on data
- storing the program and the data both into a common linear addressable memory
- executing the instructions sequentially and having instructions to alter the sequence of executed instructions

These guiding principles apply to a series of computers, and with the addition of various other inventions, they are still true for most of the processors built today. Not only did research institutes start developing computers, but commercial companies did as well. Still, each of the early computers was a unique piece.

1.1. ISA

At the beginning of the computer age, it was believed that the world would need not more than five computers. But this was soon shown to be wrong. The development advanced fast but even between different models of the same manufacturer there was no compatibility. Honeywell Bull used emulation techniques to have code written for an older model running on the new computer.

IBM went another way. To protect growing investment of their customers into software, an instruction set architecture 'ISA' was developed. Those architectures described the behavior and binary layout of all the instructions a computer could execute. Different computers from the same manufacturer adhere to a specific ISA were called a family.

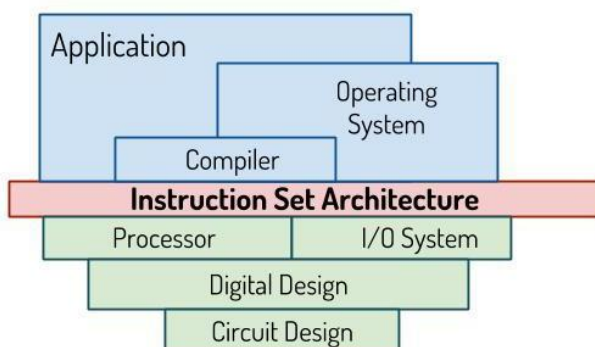


Figure 1 - ISA interface between Soft - and Hardware

Due to shortage in storage at that time, ISA was focused on creating programs with small code sizes. The number of different instructions a processor had to be able to execute became huge, putting a burden on



fast execution of the code. It also broad compatibility issues with it, as all the older instructions had still to work in a new machine.

Instruction sets became very large and complex, partly involving multiple memory accesses during the execution of a single instruction and /or switching to small code sequences (micro code) to achieve the required behavior. Today this type of instruction set is called CISC, with Intel's x86 or IBM 390 being typical representatives.

By the mid-80th a new paradigm was created: the load store approach. Here the data were first loaded from memory into registers. Operations took data only from registers and put the results back only into registers. Finally special store instructions transferred results back to memory. The RISC (Reduced Instruction Set Computer) concept was born.

A variety of companies developed their own proprietary instruction sets such as MIPS and SPARC, as well as IBM PowerPC and ARM and many more. MIPS is a commercial variation of the first RISC processor developed at the University of California at Berkeley. At one time or another some special instructions or required behavior were introduced into those instruction set, which later became a burden for subsequent implementations.

With all this in mind, the development of an open instruction set architecture, similar to open-source code, for RISC-type processors was started at the University of California, Berkeley. This project is called RISC-V, and its development and extensions are governed by the RISC-V Foundation

Processor architecture is not a very precisely defined term. Most often, it describes two elements of a processor:

- The instruction set architecture (ISA) which defines all the instructions the processor understands, their names, their layout (code point of the instructions) and all the registers the instructions can use.
- The microarchitecture of the processor which defines "how the operations for each instruction are implemented and executed in hardware.

1.2. RISC-V

RISC-V is an Instruction Set Architecture (ISA) that was created in 2011 in the PAR Lab at the University of California, Berkeley. The goal was for RISC-V to become a "Universal ISA" for processors used across the entire range of applications, from small, constrained, low-resource IoT devices to supercomputers. RISC-V architects established five principles for the architecture to achieve this goal:

- It must be compatible with a wide range of software packages and programming languages.
- Its implementation must be feasible across all technology options, from FPGAs to ASICs (application-specific integrated circuits) as well as emerging technologies.
- It must be efficient in various microarchitecture scenarios, including those implementing microcode or hardwired control, in-order or out-of-order pipelines, various types of parallelism, etc.
- It must be adaptable to specific tasks to achieve maximum performance without drawbacks imposed by the ISA itself.
- Its base instruction set must be stable and long-lasting, offering a common and solid framework for developers.



RISC-V is an open standard, in fact, the specification is public domain, and it has been managed since 2015 by the **RISC-V Foundation**, now called **RISC-V International**, a non-profit organization promoting the development of hardware and software for RISC-V architectures. In 2018, the RISC-V Foundation began an ongoing collaboration with the Linux Foundation, and in March 2020 the RISC-V Foundation became RISC-V International headquartered in Switzerland. This transition dissipated any concern the community might have had about future openness of the standard. As of 2020, RISC-V International is supported by more than 200 key players from research, academia, and industry, including Microchip, NXP, Samsung, Qualcomm, Micron, Google, Alibaba, Hitachi, Nvidia, Huawei, Western Digital, ETH Zurich, KU Leuven, UNLV, and UCM.

RISC-V is one of the few, and probably the only, globally relevant ISAs created in the past 10-20 years because of it being an open standard and modular, instead of incremental. Its modularity makes it both flexible and sleek. Processors implement the base ISA and only those extensions that are used. This modular approach differs from traditional ISAs, such as x86 or ARM, that have incremental architectures, where previous ISAs are expanded and each new processor must implement all instructions, even those that are tagged as “obsolete”, to ensure compatibility with older software programs. As an example, x86, that started with 80 instructions, has now over 1300, or 3600 if you consider all different opcodes available in machine code. This large number of instructions and the requirement of backward compatibility result in large, power-hungry processors that must support long instructions, because most of the short opcodes, or small instructions, are already in use.

RISC-V has four base ISA options: two 32-bit versions (integer and embedded versions, RV32I and RV32E) and 64- and 128-bit versions (RV64I and RV128I), as shown in Table . The ISA modules marked Ratified have been ratified at this time. The modules marked Frozen are not expected to change significantly before being put up for ratification. The modules marked Draft are expected to change before ratification. The ability to build small processors is a particularly key requirement for cost-, space-, and energy-constrained devices. Instruction extensions can be added on top of these base ISAs to enable specific tasks, for example floating point operations, multiplication and division, and vector operations. These specialized hardware extensions are also included in the standard and known by the compilers, so enabling the desired options in a compiler will allow for a targeted binary code generation. Each of these extensions is identified by a letter that must be added to the core ISA to represent the hardware capabilities of the implementation, as shown in Table . For example, RVM is the multiply/divide extension, RVF is the floating-point extension, and so on.

Base	Version	Status
RVWMO	2.0	Ratified
RV32I	2.1	Ratified
RV64I	2.1	Ratified
<i>RV32E</i>	<i>1.9</i>	<i>Draft</i>
<i>RV128I</i>	<i>1.7</i>	<i>Draft</i>

Table 1. RISC-V base ISAs

(table from <https://riscv.org/technical/specifications/>)



Extension	Version	Status
Zifencei	2.0	Ratified
Zicsr	2.0	Ratified
M	2.0	Ratified
<i>A</i>	<i>2.0</i>	<i>Frozen</i>
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
<i>Ztso</i>	<i>0.1</i>	<i>Frozen</i>
<i>Counters</i>	<i>2.0</i>	<i>Draft</i>
<i>L</i>	<i>0.0</i>	<i>Draft</i>
<i>B</i>	<i>0.0</i>	<i>Draft</i>
<i>J</i>	<i>0.0</i>	<i>Draft</i>
<i>T</i>	<i>0.0</i>	<i>Draft</i>
<i>P</i>	<i>0.2</i>	<i>Draft</i>
<i>V</i>	<i>0.7</i>	<i>Draft</i>
<i>N</i>	<i>1.1</i>	<i>Draft</i>
<i>Zam</i>	<i>0.1</i>	<i>Draft</i>

Table 2. RISC-V standard ISA extensions

(table from <https://riscv.org/technical/specifications/>)

The letter G, that denotes “general”, is used to denote the inclusion of all MAFD extensions. Note that a company or an individual may develop proprietary extensions using opcodes that are guaranteed to be unused in the standard modules. This allows third-party implementations to be developed in a faster time-to-market.

For example, a 64-bit RISC-V implementation, including all four general ISA extensions plus *Bit Manipulation* and *User Level Interrupts*, is referred to as an RV64GBN ISA. All these modules are covered in the unprivileged or user specification. RISC-V International also covers a set of requirements and instructions for privileged operations required for running general-purpose operating systems.

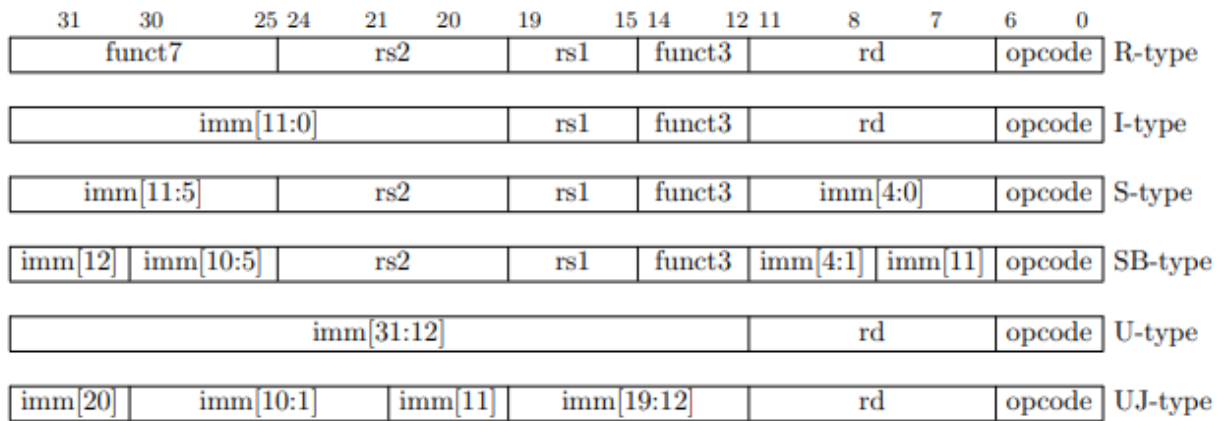
1.3. Integer ISA RV32I

RV32I (integer) is the basic instruction set that each RISC-V processor must be able to execute. It defines seventy-four 32-bit integer instructions, which act on thirty-two addressable 32-bit registers and the program counter.

The layout of instructions (the instruction format) is designed so that each element of the instruction is in a fixed position. Since not every element is needed by each instruction, positions are reused to fit everything into the given instruction size of 32 bits. However, when elements are needed, they are always in the same position, such as the opcode, sub-opcode, or register addresses. For RV32I, four instruction formats are



defined, with two of them having variations in the immediate part, resulting in a total of six formats, as defined below.



2. Processor Architecture

2.1. Single-Cycle Processor

Single-cycle processors operate using a single clock cycle per instruction, with the duration of each cycle determined by the slowest instruction's total delay. This limitation means that faster instructions cannot be executed more quickly, presenting a drawback of this approach.

2.2. Multicycle Processor

A multi-cycle processor breaks an instruction into multiple shorter steps and executes a single instruction over several clock cycles, typically without initiating a new instruction during this period.

2.3. Pipelined Processor

Pipelining is used in processors to enhance their overall performance and efficiency. In a pipelined processor, instructions are divided into multiple stages (fetch, decode, execute, etc.), allowing multiple instructions to be processed simultaneously in different stages of the pipeline. The primary reasons for using pipelining in processors include:

- **Increased Instruction Throughput:**

Pipelining allows multiple instructions to be processed simultaneously at different stages of execution. Instead of completing one instruction before starting the next, the processor begins the next instruction before the previous one finishes. This overlap increases the number of instructions that can be processed in each time period, significantly boosting the instruction throughput of the processor.

- **Improved CPU Utilization:**

In a non-pipelined processor, various parts of the CPU (like the instruction decoder, ALU, and memory access units) can be idle while waiting for other parts to complete their tasks. Pipelining ensures that all parts of the processor are actively working on different stages of multiple instructions, leading to better utilization of the CPU's resources.

- **Reduced Instruction Latency:**

While pipelining does not reduce the execution time of a single instruction, it does reduce the average time between the initiation of successive instructions. This means that although the



latency (the time it takes to complete a single instruction) remains the same, the overall instruction rate is increased, effectively reducing the time required to execute a series of instructions.

- **Higher Clock Speeds:**

By breaking down instruction processing into smaller, more manageable stages, each stage can be optimized to complete in less time. This allows the processor to operate at a higher clock speed, as the time needed for each stage to complete is minimized. Higher clock speeds mean more cycles per second, leading to faster processing.

2.4. Superscalar Processor

A superscalar processor is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor. In contrast to a scalar processor, which can execute at most one single instruction per clock cycle, a superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor.

2.5. Hazards – Data & Control

In pipelined processors, hazards are situations that can disrupt the smooth flow of instructions through the pipeline, leading to delays or incorrect results. Hazards are typically categorized into **data hazards** and **control hazards**.

2.5.1. Data Hazards

A data hazard happens when you must wait for an operand to be computed from some previous step. Data hazards in pipelining occur when there are dependencies between instructions, that is, the output of one instruction is an input to another.

They can be classified into three main types:

1. **Read After Write (RAW) Hazard:** This occurs when an instruction needs to read a value that is produced by a previous instruction that has not yet completed its writeback stage. For example, if Instruction 1 writes to a register and Instruction 2 reads from that same register, Instruction 2 must wait until Instruction 1 has completed its write operation.

Example:

ADD R1, R2, R3 ; Instruction 1: $R1 = R2 + R3$
SUB R4, R1, R5 ; Instruction 2: $R4 = R1 - R5$

Instruction 2 depends on the result of Instruction 1. If Instruction 1 hasn't finished writing to R1, Instruction 2 will have incorrect data.

2. **Write After Read (WAR) Hazard:** This occurs when an instruction writes to a register that a previous instruction has not yet read. If the write operation occurs before the read operation, it can cause the read instruction to use incorrect data.

Example:

MOV R1, R2 ; Instruction 1: $R1 = R2$
ADD R1, R3 ; Instruction 2: $R1 = R1 + R3$

If Instruction 2 writes to R1 before Instruction 1 completes its write, the final value in R1 will be incorrect.

3. **Write After Write (WAW) Hazard:** This occurs when two instructions write to the same register. The final value of the register depends on the order of execution of these instructions.



Example:

MOV R1, R2 ; Instruction 1: $R1 = R2$

ADD R1, R3 ; Instruction 2: $R1 = R1 + R3$

If Instruction 2 writes to R1 before Instruction 1 completes its write, the final value in R1 will be incorrect.

Handling Data Hazards:

- **Stalling:** Inserting pipeline stalls (or bubbles) to delay the execution of instructions until the hazard is resolved. This is done by inserting NOPs (no-operation instructions) or using special hardware mechanisms to hold up the pipeline.
- **Forwarding (or Bypassing):** Using additional hardware paths to forward the result of an instruction directly to the next instruction that needs it, bypassing the need for the result to be written back to the register file first.
- **Register Renaming:** Using additional registers to avoid conflicts between instructions that use the same register names. This helps in eliminating WAW and WAR hazards by providing unique register identifiers.

2.5.2. Control Hazards

A control hazard happens when a CPU can't tell which instructions it needs to execute next. Control hazard occurs whenever the pipeline makes incorrect branch prediction decisions, resulting in instructions entering the pipeline that must be discarded. A control hazard is often referred to as a branch hazard.

Types of Control Hazards:

1. **Branch Hazard:** Occurs when a branch instruction changes the execution path, causing subsequent instructions to be incorrect or invalid. This hazard arises because the pipeline may have already fetched instructions that are not needed.
2. **Jump Hazard:** Similar to branch hazards but involves unconditional jumps to different locations in the code.

Handling Control Hazards:

- **Branch Prediction:** Predicting the outcome of a branch instruction to keep the pipeline filled with the likely correct instructions. There are two main types:
 - **Static Prediction:** Uses a fixed method to predict the outcome, such as always predicting a branch will not be taken.
 - **Dynamic Prediction:** Uses historical execution patterns and hardware mechanisms to make predictions about branch behavior.
- **Branch Target Buffer (BTB):** A cache that stores the addresses of recently executed branches and their target addresses, helping in quicker branch resolution.

- Delayed Branch: Reordering instructions to fill the delay slots with instructions that can be executed regardless of the branch outcome. This approach makes use of the cycles that would otherwise be wasted.
- Pipeline Flushing: When a branch instruction is resolved, any incorrectly fetched instructions are discarded, and the correct instructions are fetched and executed. This involves flushing the pipeline and starting over from the correct address.

3. SweRV EH1

The **SweRV EH1** is a RISC-V processor core developed by Western Digital. It is a superscalar core with a dual-issue, 9-stage pipeline (see Figure 2) that supports four arithmetic logic units (ALUs), labeled EX1 to EX4, across two pipelines, I0 and I1. Both pipelines support ALU operations. One pipeline handles load/store operations, while the other pipeline includes a multiplier with a 3-cycle latency. **Additionally, the processor has an out-of-pipeline divider with a 34-cycle latency.**

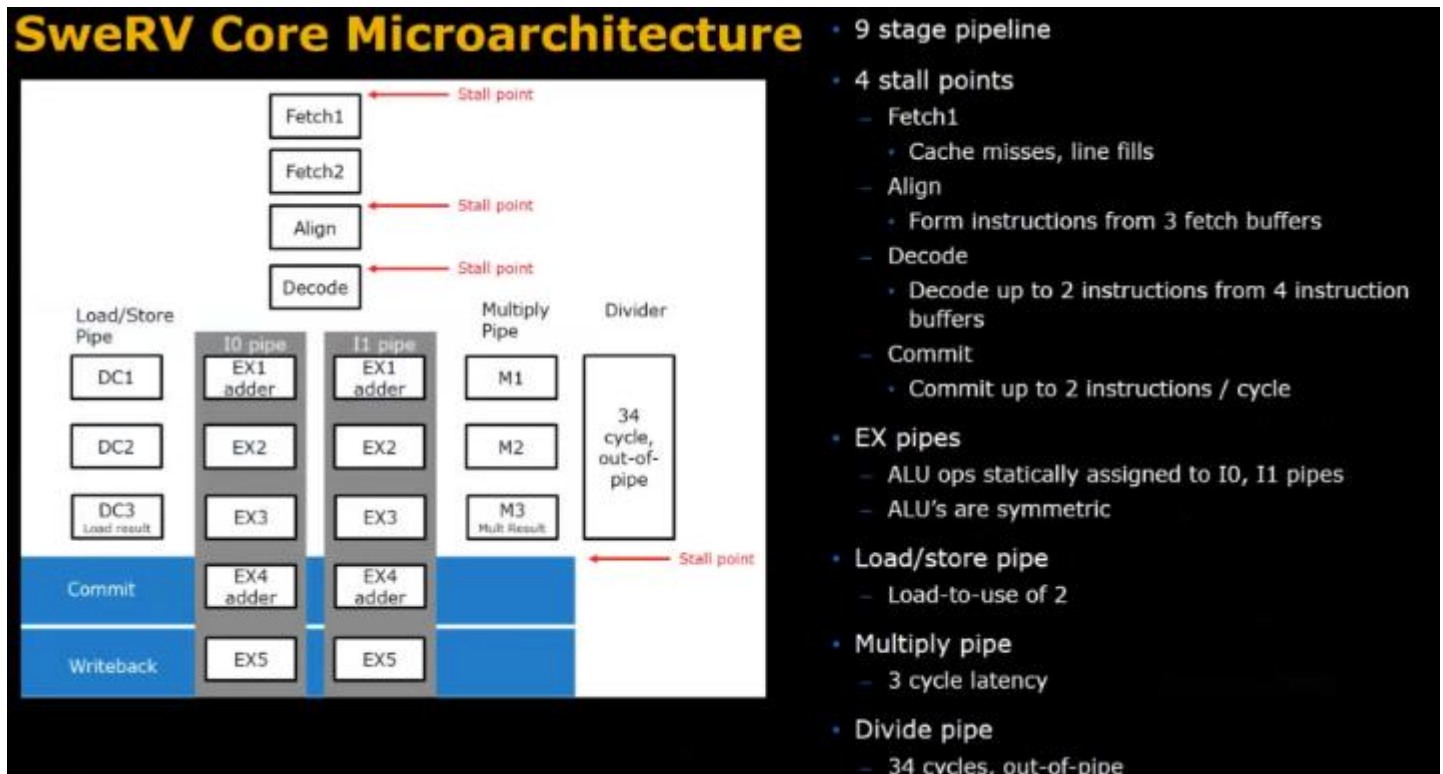


Figure 2. SweRV EH1 core microarchitecture

The SweRV EH1 processor pipes:

I0/I1 Pipes:

- Two integer pipes which have three stages (EX1, EX2, and EX3).
- EX1 performs the ALU operation.

Multiply Pipe:

- The multiply pipe contains a 3-cycle integer multiplier using three stages (M1, M2, and M3).

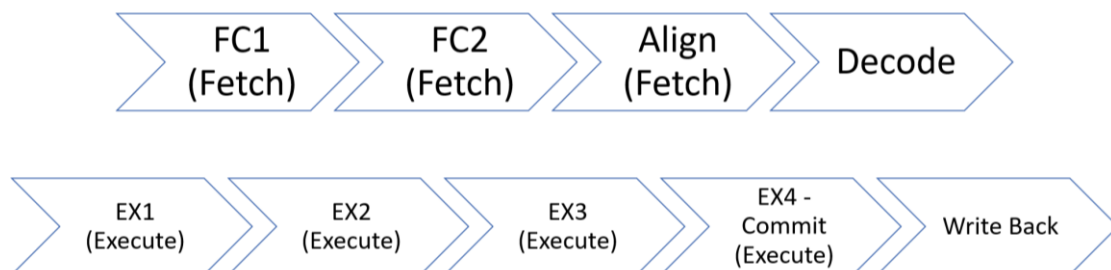
Load/Store (L/S) Pipe:

- The L/S pipe executes both load and store instructions.
- DC1: an adder calculates the address by adding the register base address and the offset

Divider:

- The divider is a non-pipelined 34-cycle integer divider.

Pipeline stages:



1. **Fetch** stages - read instructions from the Instruction Memory
 - a. **FC1**: Computes the instruction address (ifc_fetch_addr_f1)
 - b. **FC2**: Reads 128-bit instruction bundles from the I\$ (caching memory within the main memory address range) or the DDR (external memory).
2. **Align** performs two main tasks:
 - Provides two 32-bit instructions per cycle to the Decode stage: Extracts two instructions per cycle from the 128-bit bundles provided by the Instruction Memory and assigns them to each of the two ways available in SweRV EH1.
 - Uncompresses 16-bit instruction into 32-bit instruction.
3. **Decode** performs two main tasks:
 - Decode the instructions and generate the control signals (performed by the Control Unit).
 - Distribute the instructions and operands to the appropriate pipes:

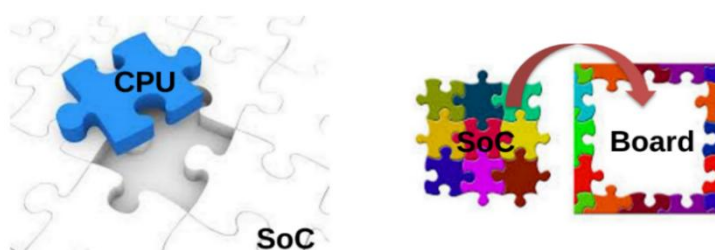
- Two Integer pipes: I0 and I1
 - Multiply pipe
 - Load/Store pipe (L/S)
 - Out-of-pipe 34-cycle Divide
4. **EX1**: Performs the ALU operation
 5. **EX2**: Execute stage 2 (for multicycle instructions)
Multicycle instructions are instructions that require more than one clock cycle to complete due to their complexity.
 6. **EX3**: Execute stage 3 (for multicycle instructions)
 7. **EX4/Commit**: Selects the result to write back to the register file
 8. **Write Back**: Writes the results to the Register File using write ports 0 and 1. The Control Pipeline Registers supply the register identifiers and the enable signals (which were generated in the Decode stage).

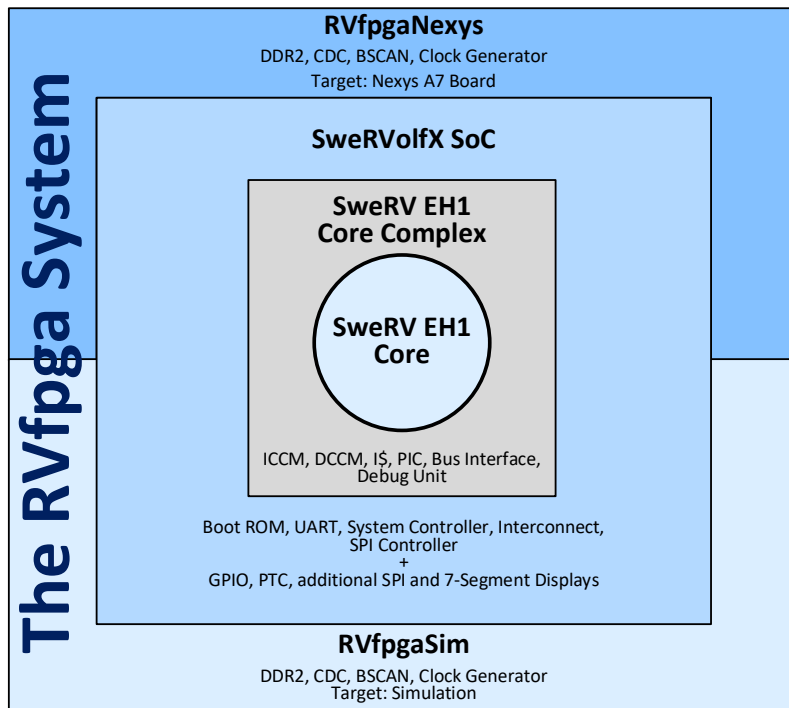
At the end of the third execution stage (EX3/DC3/M3), the result of the instructions is selected from the proper pipe (I0/I1, MUL, or L/S) using two 3:1 multiplexers, one for each way. The Divider has its own path to the Register File.

Four stall points (delays in the processor pipeline caused by dependencies or resource conflicts) exist in the pipeline: 'Fetch 1', 'Align', 'Decode', and 'Commit'. The 'Fetch 1' stage includes a Gshare branch predictor. In the 'Align' stage, instructions are retrieved from three fetch buffers. In the 'Decode' stage, up to two instructions from four instruction buffers are decoded. In the 'Commit' stage, up to two instructions per cycle are committed. Finally, in the 'Writeback' stage, the architectural registers are updated.

4. RVFPGA SYSTEM OVERVIEW

RISC-V FPGA, also written RVfpga, is a package that includes instructions, tools, and labs for targeting a commercial RISC-V processor to a field programmable gate array (FPGA) and to a simulator, and then using and expanding it to learn about computer architecture, digital design, embedded systems, and programming.





SweRV Core

- Open-source core from Western Digital
- 2-way superscalar core
- 9-stage pipeline
- In-order
- RV32IMC

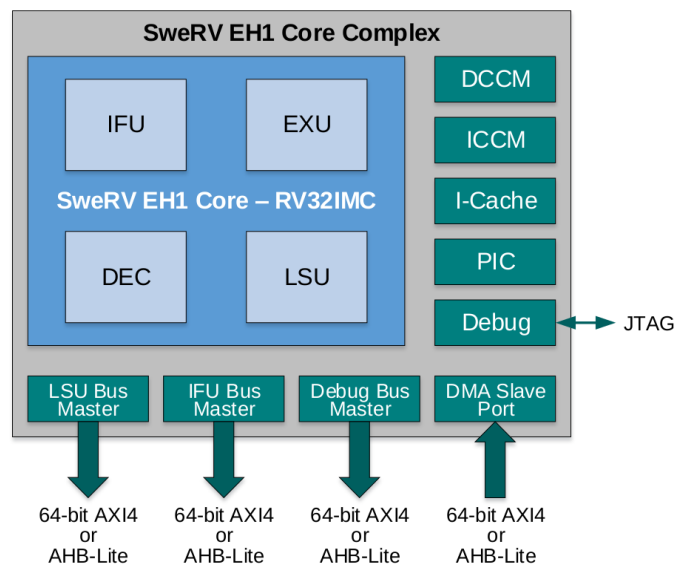
Figure 3. RVfpga System Hierarchy

<u>SweRV EH1 Core</u>	<p>Open-source commercial RISC-V core developed by Western Digital.</p> <p>SweRV EH1 Core is a 32-bit CPU core which supports RISC-V's integer (I), compressed instruction (C), and integer multiplication and division (M) extensions.</p>
	<p>SweRV EH1 Core – RV32IMC</p> <p>IFU: Instruction Fetch Unit</p> <p>EXU: Execution Unit</p> <p>DEC: Decode Unit</p> <p>LSU: Load Store Unit</p>

SweRV EH1 Core Complex

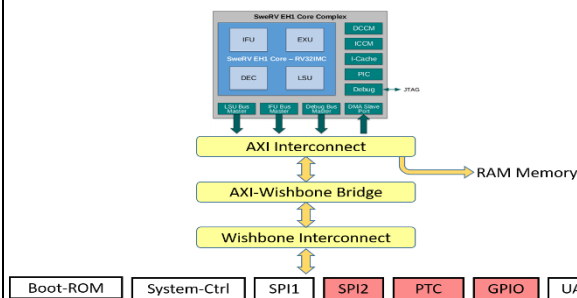
An extension to the SweRV EH1 Core which adds the following elements to the EH1 Core described above:

- Two dedicated memories, one for instructions (ICCM) and the other for data (DCCM), which are tightly coupled to the core. These memories provide low-latency access and SECDED ECC (single-error correction and double-error detection error correcting codes) protection. Each of the memories can be configured as 4, 8, 16, 32, 48, 64, 128, 256, or 512KB.
- An optional 4-way set-associative instruction cache with parity or ECC protection.
- An optional Programmable Interrupt Controller (PIC), that supports up to 255 external interrupts.
- Four system bus interfaces for instruction fetch (IFU Bus Master), data accesses (LSU Bus Master), debug accesses (Debug Bus Master), and external DMA accesses (DMA Slave Port) to closely coupled memories (configurable as 64-bit AXI4 or AHB-Lite buses).
- Core Debug Unit compliant with the RISC-V Debug specification.



SweRVolfX SoC

- Open-source system-on-chip (SoC) from Chips Alliance
- SweRVolf uses the SweRV EH1 Core. SweRVolf includes a Boot ROM, UART, and a System Controller and an SPI controller (SPI1)
- SweRVolfX extends SweRVolf with another SPI controller (SPI2), a GPIO (General Purpose Input/Output), 8-digit 7-Segment Displays and a PTC (shown in red).
- SweRV EH1 Core uses an AXI bus and peripherals use a Wishbone bus, so the SoC also has an AXI to Wishbone Bridge



SweRVolfX Memory Map

System	Address
Boot ROM	0x80000000 - 0x80000FFF
System Controller	0x80001000 - 0x8000103F
SPI1	0x80001040 - 0x8000107F
SPI2	0x80001100 - 0x8000113F
Timer	0x80001200 - 0x8000123F
GPIO	0x80001400 - 0x8000143F
UART	0x80002000 - 0x80002FFF

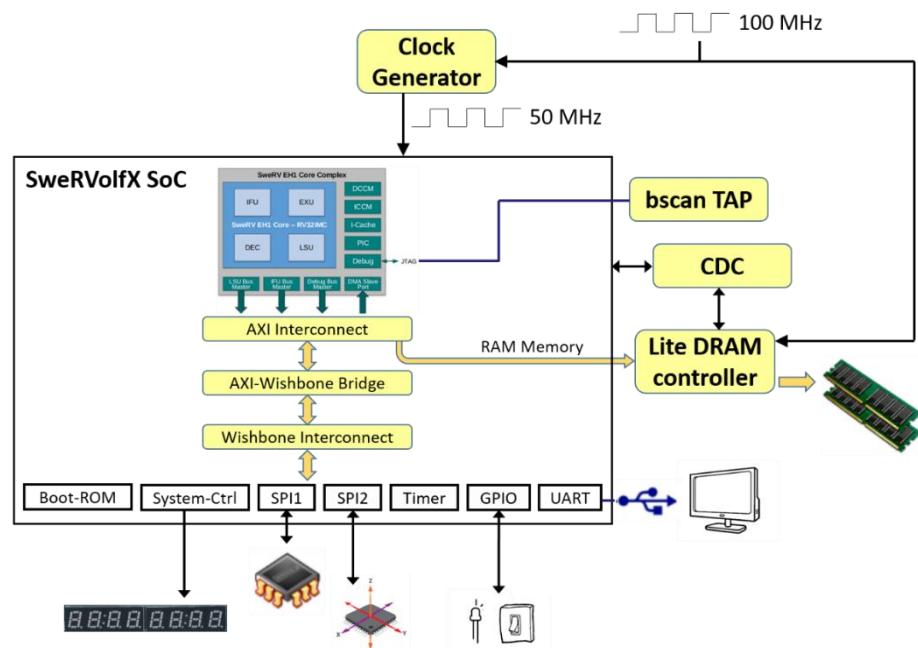
RVfpgaNexys

The SweRVolfX SoC targeted to the Nexys A7 board and its peripherals. It adds a DDR2 interface, CDC (clock domain crossing) unit, BSCAN logic (for the JTAG interface), and clock generator.

RVfpgaNexys is the same as SweRVolf Nexys

(<https://github.com/chipsalliance/Cores-SweRVolf>), except that the latter is based on SweRVolf.

- Memory/Peripherals used in RVfpgaNexys from the Nexys A7 (or Nexys4 DDR) FPGA board:
 - **DDR2 memory** (accessed through the Lite DRAM controller mentioned above)
 - **USB connection**
 - **SPI Flash memory**
 - **SPI Accelerometer**
 - **16 LEDs and 16 Switches**
 - **8-digit 7-Segment Displays**



RVfpgaSim

RVfpgaSim is the SweRVolfX SoC wrapped in a testbench to be used by HDL simulators.



5. RISC-V Assembly Instructions and code examples

5.1. RISC-V Assembly Instructions

RISC-V Assembly	Description	Operation
add s0, s1, s2	Add	$s0 = s1 + s2$
sub s0, s1, s2	Subtract	$s0 = s1 - s2$
addi t3, t1, -10	Add immediate	$t3 = t1 - 10$
mul t0, t2, t3	32-bit multiply	$t0 = t2 * t3$
div t9, t5, t6	Division	$t9 = t5 / t6$
rem s4, s1, s2	Remainder	$s4 = s1 \% s2$
and t0, t1, t2	Bit-wise AND	$t0 = t1 \& t2$
or t0, t1, t5	Bit-wise OR	$t0 = t1 t5$
xor s3, s4, s5	Bit-wise XOR	$s3 = s4 \wedge s5$
andi t1, t2, 0xFFB	Bit-wise AND immediate	$t1 = t2 \& 0xFFFFFBB$
ori t0, t1, 0x2C	Bit-wise OR immediate	$t0 = t1 0x2C$
xori s3, s4, 0xABC	Bit-wise XOR immediate	$s3 = s4 \wedge 0xFFFFFABC$
sll t0, t1, t2	Shift left logical	$t0 = t1 \ll t2$
srl t0, t1, t5	Shift right logical	$t0 = t1 \gg t5$
sra s3, s4, s5	Shift right arithmetic	$s3 = s4 \ggg s5$
slli t1, t2, 30	Shift left logical immediate	$t1 = t2 \ll 30$
srli t0, t1, 5	Shift right logical immediate	$t0 = t1 \gg 5$
srai s3, s4, 31	Shift right arithmetic immediate	$s3 = s4 \ggg 31$
lw s7, 0x2C(t1)	Load word	$s7 = \text{memory}[t1+0x2C]$
lh s5, 0x5A(s3)	Load half-word	$s5 = \text{SignExt}(\text{memory}[s3+0x5A]_{15:0})$
lb s1, -3(t4)	Load byte	$s1 = \text{SignExt}(\text{memory}[t4-3]_{7:0})$
sw t2, 0x7C(t1)	Store word	$\text{memory}[t1+0x7C] = t2$
sh t3, 22(s3)	Store half-word	$\text{memory}[s3+22]_{15:0} = t3_{15:0}$
sb t4, 5(s4)	Store byte	$\text{memory}[s4+5]_{7:0} = t4_{7:0}$
beq s1, s2, L1	Branch if equal	if $(s1 == s2)$, PC = L1
bne t3, t4, Loop	Branch if not equal	if $(s1 \neq s2)$, PC = Loop
blt t4, t5, L3	Branch if less than	if $(t4 < t5)$, PC = L3
bge s8, s9, Done	Branch if not equal	if $(s8 \geq s9)$, PC = Done
li s1, 0xABCDEF12	Load immediate	$s1 = 0xABCDEF12$
la s1, A	Load address	$s1 = \text{Variable A's memory address (location)}$



nop	Nop	no operation
mv s3, s7	Move	s3 = s7
not t1, t2	Not (Invert)	t1 = ~t2
neg s1, s3	Negate	s1 = -s3
j Label	Jump	PC = Label
jal L7	Jump and link	PC = L7; ra = PC + 4
jr s1	Jump register	PC = s1

RISC-V 32-bit Registers:

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-t2	x5-7	Temporary variables
s0/fp	x8	Saved variable / Frame pointer
s1	x9	Saved variable
a0-1	x10-11	Function arguments / Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved variables
t3-t6	x28-31	Temporary variables

5.2. RISC-V main directives

Directive	Description
.text	Subsequent items are stored in the text section (machine code).
.data	Subsequent items are stored in the data section (global variables).
.bss	Subsequent items are stored in the bss section (global variables initialized to 0).
.section .foo	Subsequent items are stored in the section named .foo.
.align n	Align the next datum on a 2 ⁿ -byte boundary. For example, .align 2 aligns the next value on a word boundary.
.balign n	Align the next datum on an n-byte boundary. For example, .balign 4



	aligns the next value on a word boundary.
.globl sym	Declare that label sym is global and may be referenced from other files
.string "str"	Store the string str in memory and null-terminate it.
.word w1,...,wn	Store the n 32-bit quantities in successive memory words.
.byte b1,...,bn	Store the n 8-bit quantities in successive bytes of memory.
.space	Reserve memory space to store variables without an initial value. It is commonly used to declare the output variables, when they are not also serving as input variables. The space we want to reserve must always be expressed as a number of bytes. For example, the directive RES: .space 4 reserves four bytes (i.e. one word) that are not initialized.
.equ name,constant	Define symbol name with value constant. For example, .equ N,12, defines symbol N with the value 12.
.end	The assembler will conclude its work when it reaches the directive .end. Any text located after this directive will be ignored.
	Description
.text	Subsequent items are stored in the text section (machine code).
.data	Subsequent items are stored in the data section (global variables).
.bss	Subsequent items are stored in the bss section (global variables initialized to 0).
.section .foo	Subsequent items are stored in the section named .foo.
.align n	Align the next datum on a 2n-byte boundary. For example, .align 2 aligns the next value on a word boundary.
.balign n	Align the next datum on an n-byte boundary. For example, .balign 4 aligns the next value on a word boundary.
.globl sym	Declare that label sym is global and may be referenced from other files
.string "str"	Store the string str in memory and null-terminate it.



.word w1,...,wn	Store the n 32-bit quantities in successive memory words.
.byte b1,...,bn	Store the n 8-bit quantities in successive bytes of memory.

5.3. RISC-V Code Examples

The examples below show how to code some common high-level constructs in RISC-V assembly. Note that branch instructions (beq, bne, blt, and bge) conditionally jump to a label, whereas the jump instruction (j) unconditionally jumps to a label. Single-line comments are indicated by `//` in C and `#` in RISC-V assembly.

In the first example notice that the C code and RISC-V assembly code check for the opposite cases: the C code checks for less than (<) and the assembly equivalent checks for greater than or equal (>=).

RISC-V Assembly Example 1- if/else statement:

// C Code int a, b, c; if (a < b) c = 5; else c = a + b;	# RISC-V Assembly # s0 = a, s1 = b, s2 = c bge s0, s1, L1 # if (a >= b) goto L1 addi s2, zero, 5 # c = 5 j L2 # jump over else block L1: add s2, s0, s1 # c = a + b L2:
--	--

In the second example, the RISC-V assembly code uses temporary registers (t0-t3) to hold temporary values, such as the constant 100 and the base address of the data array. After initializing the registers in the first three instructions, the RISC-V assembly code checks for $i \geq 100$ using the bge (branch if greater than or equal to) instruction; again, this is the opposite case from the C code. If that condition is met, the for loop is done. If the branch is **not** taken, i is less than 100 and the remaining code is executed. Notice that the index i is multiplied by 4 (using the slli t2, s0, 2 instruction) before it is added to the base address because integers (32-bit two's complement numbers) occupy 4 bytes of memory. In RISC-V, memory is byte-addressable (i.e., each byte has its own address). If the array had been an array of characters (i.e., `char data[100];`), then each array element would only occupy a byte and i could be added directly to the base address to form the address of array index i , i.e., `array[i]`. After the array element is read, decremented by ten, and written (via the lw, addi, and sw instructions, respectively), the array index i (i.e., s0) is incremented and the program jumps back to the beginning of the for loop (using the j L5 instruction).



RISC-V Assembly Example 2 - manipulating an array of integers:

// C Code	# RISC-V Assembly
	# s0 = i, t1 = base address of data (assumed # to be at 0x300)
int i;	addi s0, zero, 0 # i = 0
	addi t0, zero, 100 # t0 = 100
	li t1, 0x300 # base address of array
int data[100];	L5: bge s0, t0, L7 # if (i>=100) exit loop
	slli t2, s0, 2 # t2 = i*4
	add t2, t1, t2 # address of data[i]
for (i=0; i<100; i++)	lw t3, 0(t2) # t3 = array[i]
	addi t3, t3, -10 # t3 = array[i]-10
array[i] = array[i]-10;	sw t3, 0(t2) # array[i] = array[i]-10
	addi s0, s0, 1 # i++
	j L5 # loop
	L7:

6. Implementation environment for the experiment

We will use the Nexys A7 board as FPGA development board and Segger J-Link as debug probe.

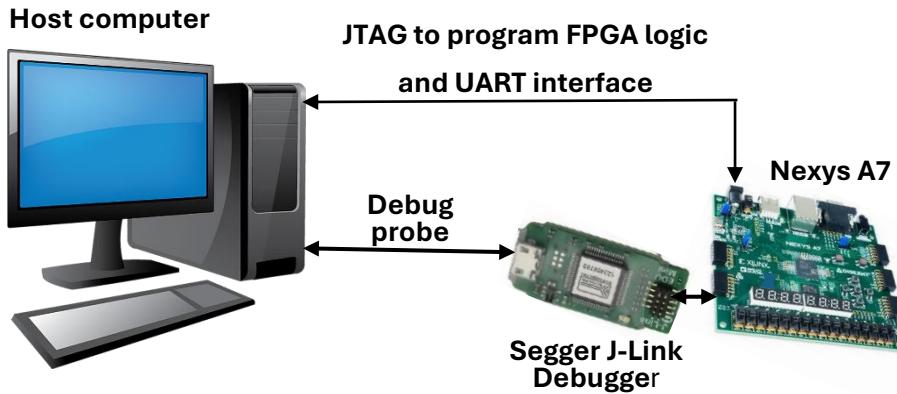
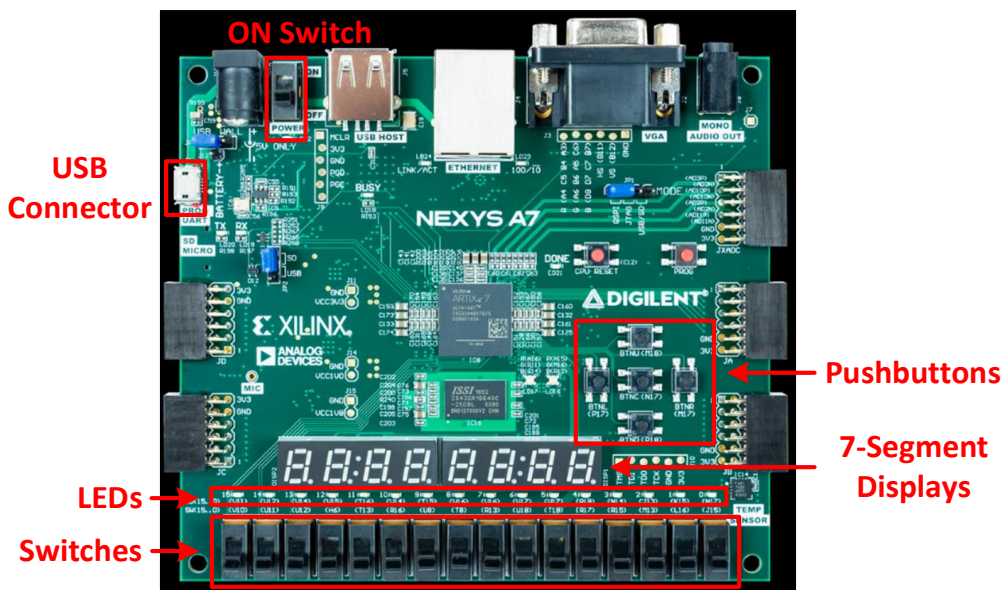


Figure 4: Experiment Setup

6.1. Digilent's Nexys A7 FPGA board



The Nexys A7-100T FPGA board includes the following interfaces and devices:

- 128 MiB DDR RAM
- 128 Mibit SPI Flash Memory
- 8-digit 7-Segment Displays
- 16 Switches

- 16 LEDs
- Sensors and connectors, including a microphone, audio jack, VGA 25 port, USB host port, RGB-LEDs, I2C temperature sensor, SPI accelerometer, among other.
- Xilinx Artix-7 FPGA, which has the following features:
 - 15.850 Logic slices of four 6-input LUTs and 8 flip-flops.
 - 4.860 Kibits of total block RAM
 - 6 clock management tiles (CMTs)
 - 170 I/O pins
 - 450 MHz internal clock frequency

6.2. Development tools

- 6.2.1. Vivado - is a Xilinx tool for viewing, modifying, synthesizing the Verilog code for RISC-V FPGA and programming the FPGA. It will be used for synthesizing the SweRVofX system and programming the FPGA
- 6.2.2. Modelsim – is a Siemens (previously Mentor Graphics) environment for simulation of hardware description languages such as VHDL, Verilog and System Verilog.
- 6.2.3. Segger Embedded Studio for RISC-V – is a development environment for devices based on the open RISC-V architecture. It will be used for developing and debugging software applications using Segger J-Link Debugger.



7. Meeting #1 – preparation report

- 7.1. How many pipeline stages does the SweRV EH1 processor have?
- 7.2. What is the maximum number of commands processed by the commit step in each cycle?
- 7.3. Provide examples of 4 different types of instruction formats as they would be decoded in the decode stage.
- 7.4. How many cycles does it take to perform integer multiplication?
- 7.5. How many cycles does it take to perform division?
- 7.6. What is the difference between **Stall** and **NOP**?
- 7.7. How many stall points are there in the pipeline of the SweRV EH1 processor?
- 7.8. Explain the difference between data hazards and control hazards.
- 7.9. Write a C program to compute the greatest common divisor (GCD) of two numbers, a and b, using the Euclidean algorithm. The values of a and b should be statically defined variables in the program. Use loops only and keep the program simple without using recursion or functions. Name the program GCD.c.

Here is some additional information about the Euclidean algorithm:
<https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm..>
- 7.10. Convert the code you wrote in 7.9 to SweRV EH1 assembly code.



8. Meeting #1 - performing the experiment in the lab

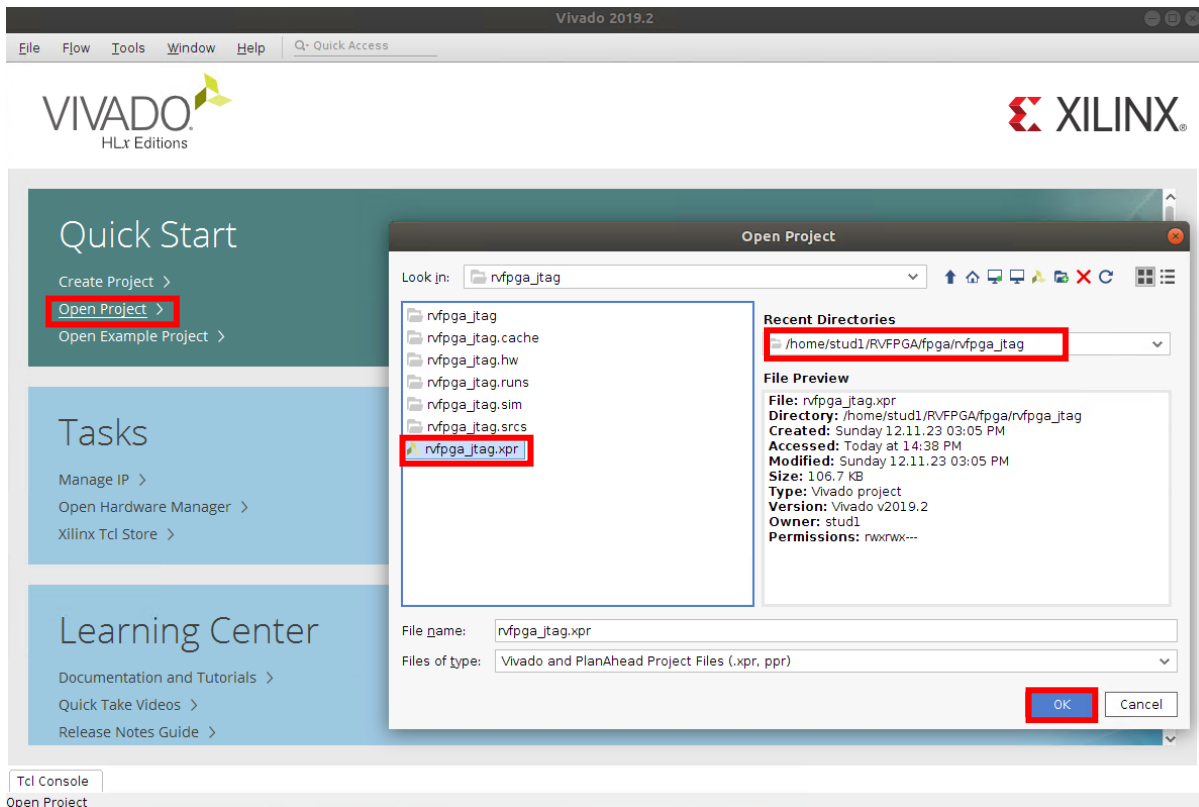
8.1. Programming the FPGA with RISC-V software

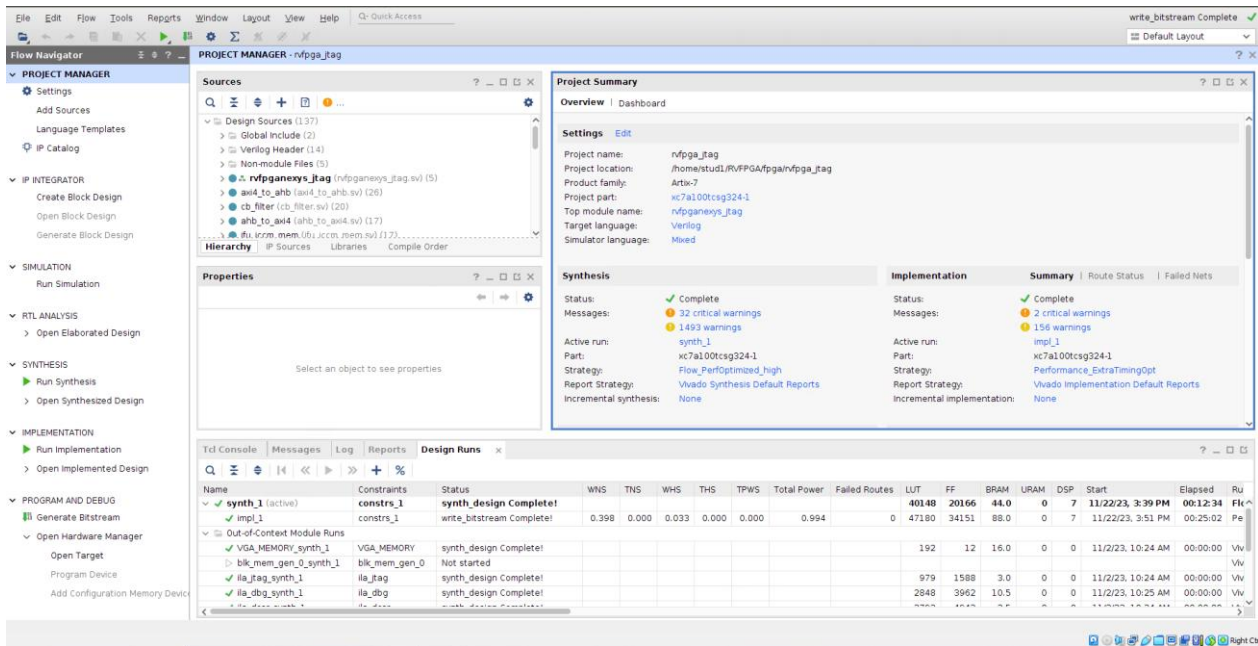
Setting up the environment & Programming the FPGA with RISC-V software using Vivado:

- Get a username and password from the lab instructor
- Login station
- Open terminal : mouse right click -> **Open Terminal**
- Change directory to ~/RVFPGA

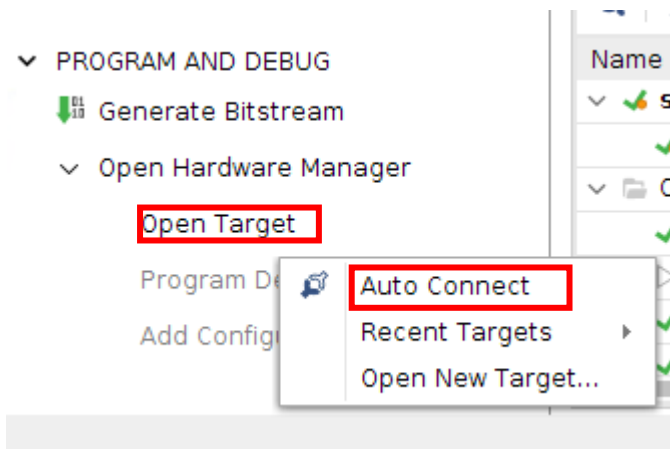
```
stud1@sysarch: ~/RVFPGA
File Edit View Search Terminal Help
stud1@sysarch:~$ cd RVFPGA/
stud1@sysarch:~/RVFPGA$
```

- Turn on the Nexys A7 board using the switch at the top left
- On the terminal run **./vivado &**
- Open the **~/RVFPGA/rvfpga_jtag/rvfpga_jtag.xpr** project in **Vivado**



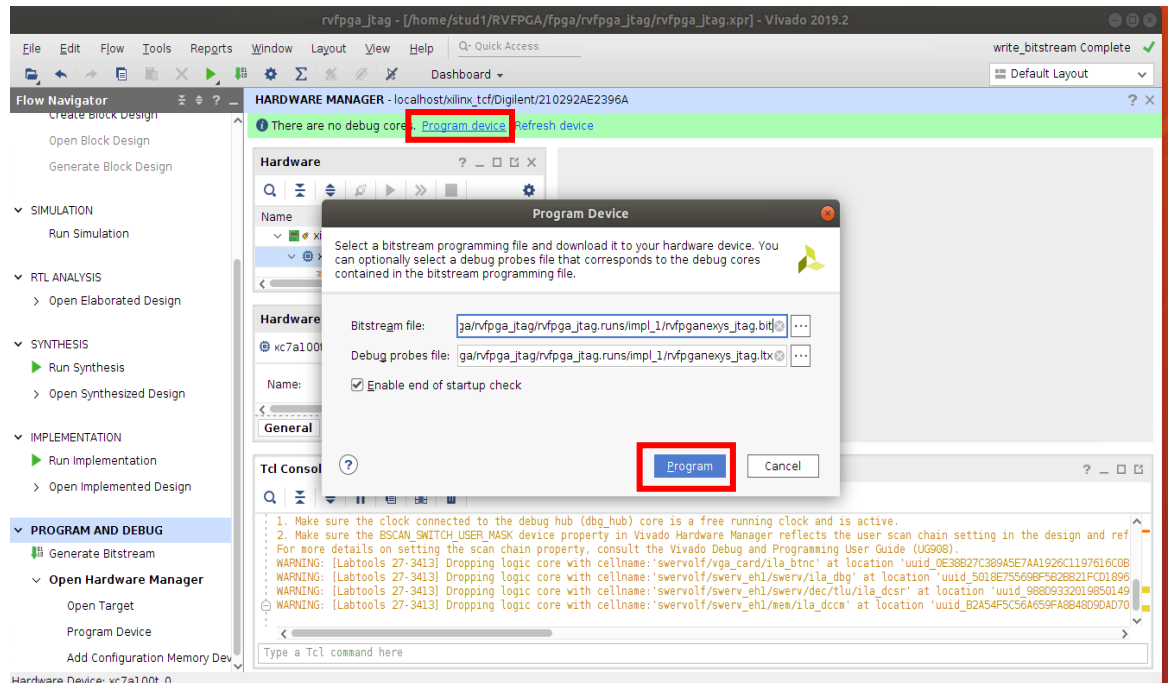


- Open the target by clicking on **Open target** (under Open Hardware Manager) and choose Auto connect:



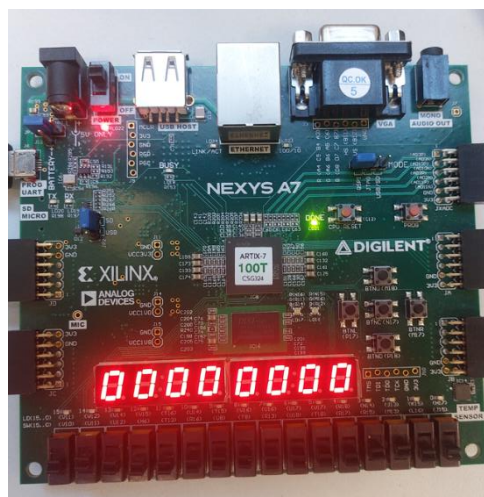


- Load RVfpgaNexys onto the FPGA:
Select **Program device** and click on **Program** as shown in:



Now the FPGA is programmed with RVfpgaNexys design :

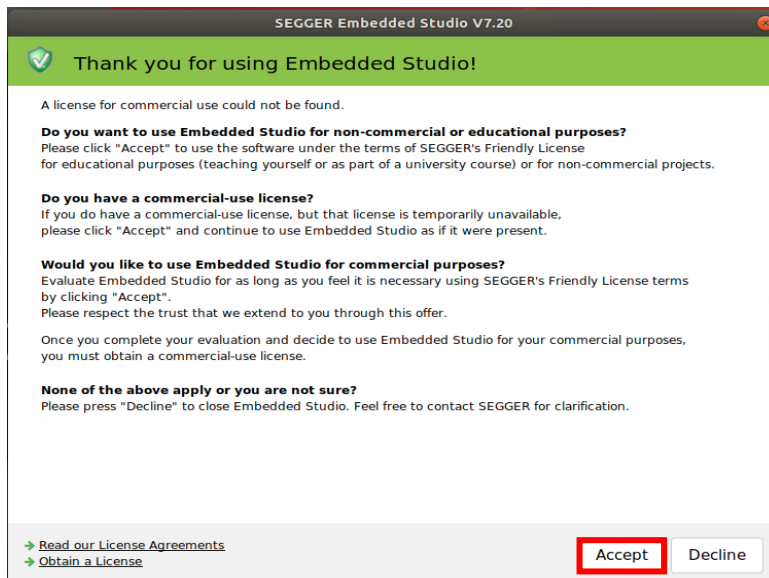
Hardware	
Name	Status
localhost (1)	Connected
xilinx_tcf/Digilent/210292AE20	Open
xc7a100t_0 (5)	Programmed
XADC (System Monitor)	
hw_ila_1 (swervolf/swerv_e	Idle
hw_ila_2 (swervolf/swerv_e	Idle
hw_ila_3 (swervolf/swerv_e	Idle
hw_ila_4 (swervolf/vga_car	Idle



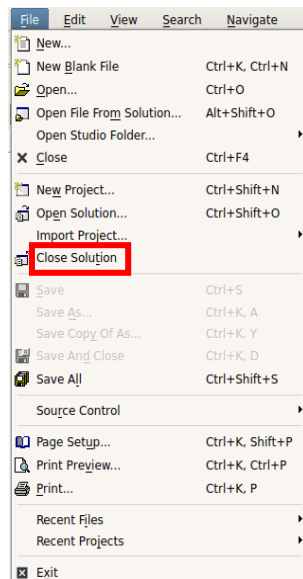
8.2. SEGGER Embedded Studio – compilation & debugging

The next step is to run and debug an assembly program on RISC-V.

- In the terminal run: **segger &**
- Click on Accept:



- Click on **File -> Close Solution**



The first example program we will run, **AL_Operations.s**, is an assembly program that performs three arithmetic-logic operations (addition, subtraction, and logical AND) on the same register, t3 (also known as x28), within an infinite loop.



```
.globl main  
main:
```

```
# Register t3 is also called register 28 (x28)
```

```
li t3, 0x0      # t3 = 0
```

```
REPEAT:
```

```
addi t3, t3, 6    # t3 = t3 + 6
```

```
addi t3, t3, -1   # t3 = t3 - 1
```

```
andi t3, t3, 3    # t3 = t3 AND 3
```

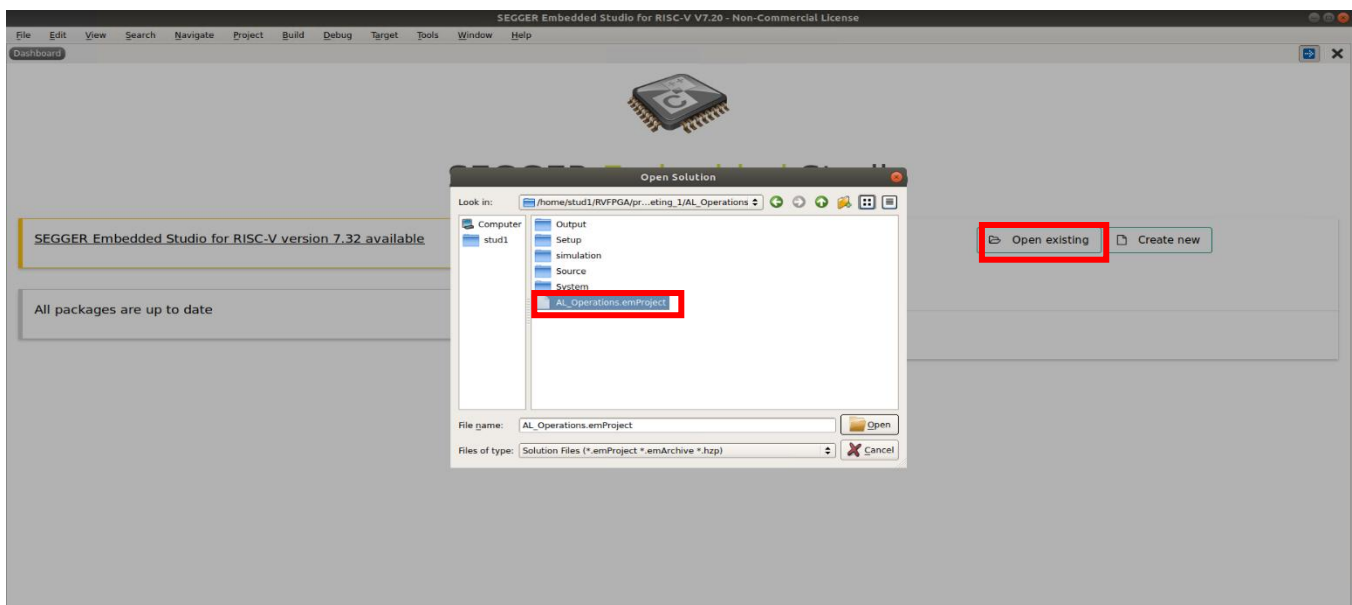
```
beq zero, zero, REPEAT # Repeat the loop
```

```
nop
```

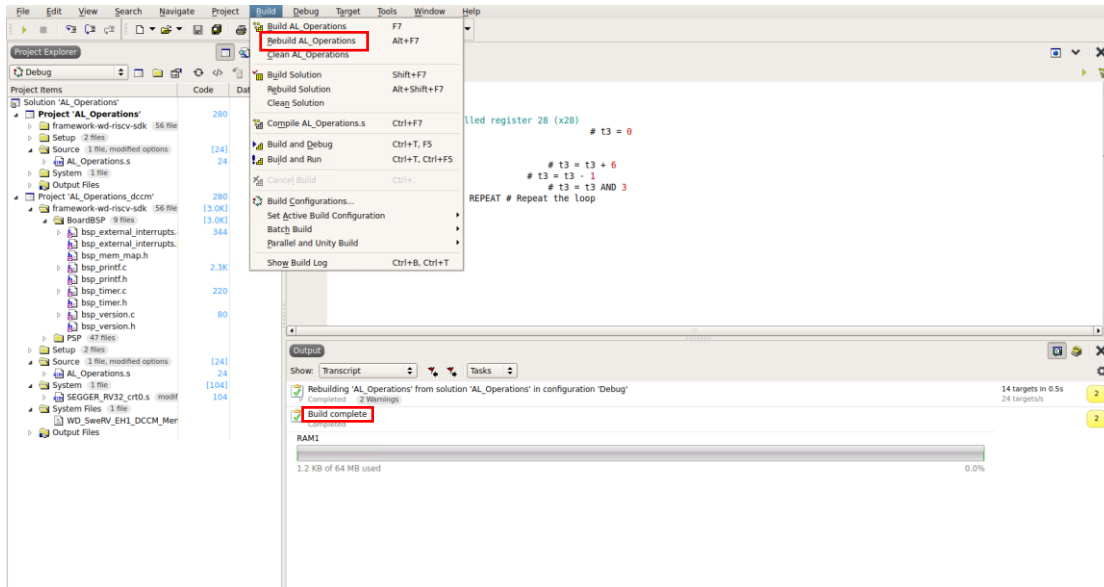
```
.end
```

In the following steps we will learn how to run and debug this code on the Nexys A7 FPGA board using Segger Embedded Studio.

- Open the assembly program, AL_Operations.S, by clicking on **Open existing** and browse to **~/RVFPGA/programs/meeting_1/AL_Operations** :



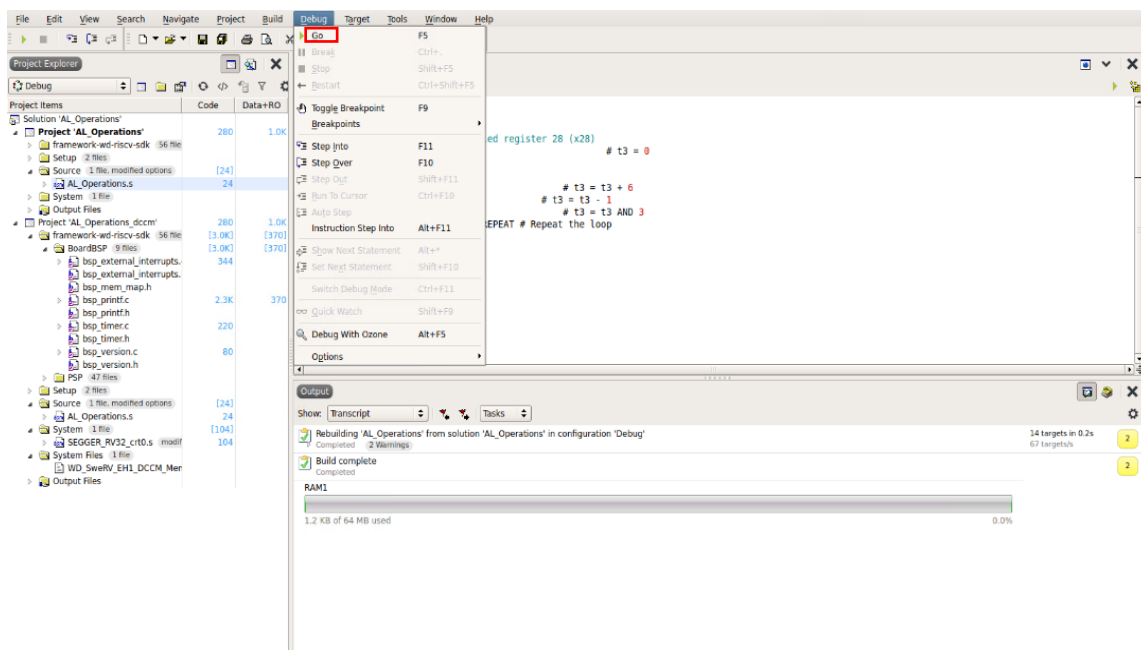
- Select: **Build-> Rebuild AL_operations**



Answer Question #1

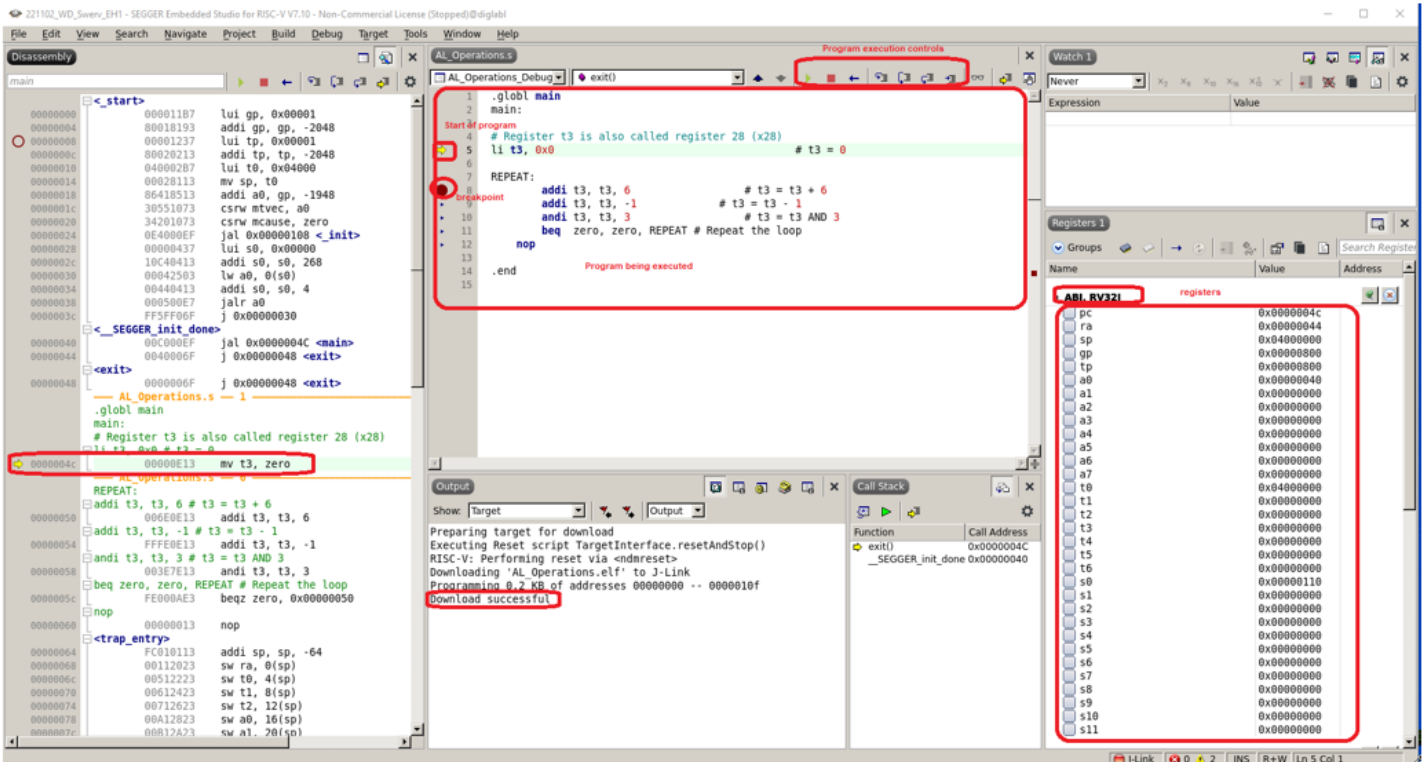
- Run and debug the Program on Nexys A7 FPGA board :

Select: Debug -> Go



If you receive an error, ensure that the J-Link is connected by using the command:

target->connect j-link



Breakpoints: Click on the desired line- red point will appear



Program execution:

- Run the program
- Stop the program
- Restart
- Step into
- Step over
- Step out

Follow the Step over and write down the actual values of t3 (Registers window) each step in the table.

Answer question #2



Change the AL_Operations.S assembly program so that it will run the GCD assembly code you wrote in the preparation.

Run& test the code on Nexys A7 FPGA board.

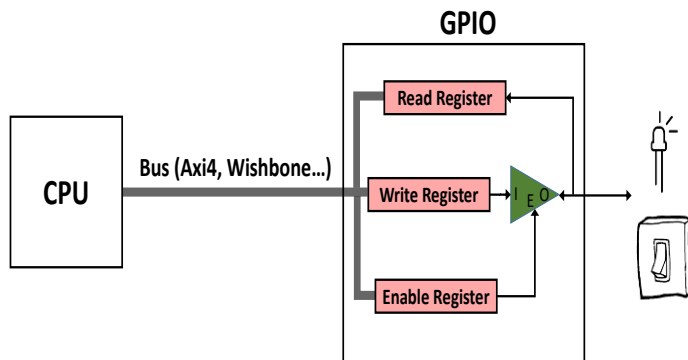
Show the instructor.

8.3. General-Purpose I/O (GPIO)

- GPIO Allows processor to read/write pins connected to peripherals (like switches and LEDs)
- Each pin can be configured as an input or output using tri-state

Three memory-mapped registers:

- **Read Register:** value read from pin
- **Write Register:** value to write to pin
- **Enable Register:** 1 = output, 0 = input



Register	Memory-Mapped Address
Read Register	0x80001400
Write Register	0x80001404
Enable Register	0x80001408

Mapping LEDs & Switches to GPIO pins:

LEDs: pins [15:0] (outputs of processor)

Switches: pins [31:16] (inputs to processor)

Configure GPIO:

Enable Register = 0x0000FFFF (1 = output, 0 = input)

```
li t0, 0x80001400
```

```
li t1, 0xFFFF
```

```
sw t1, 8(t0)      # Enable Register = 0x0000FFFF
```

Write LEDs:

Write value in [15:0] to address 0x80001404

```
sw t3, 4(t0) # LEDs = [t3]15:0
```

Read Switches:

Read switches in bits [31:16] from address 0x80001400

Shift right by 16 bits to put value in lower 16 bits

```
lw t5, 0(t0) # [t5]31:16 = switch values
```

```
srli t5, t5, 16 # [t5]15:0 = switch values
```



Change the **AL_Operations.S** assembly program so that it will continuously loop, reading input from the switches and writing the values to the LEDs.

Run& test the code on Nexys A7 FPGA board.

Answer Question #3

In AL_Operations.S delete the code you wrote and restore the previous code:

```
.globl main  
main:
```

```
# Register t3 is also called register 28 (x28)  
li t3, 0x0      # t3 = 0
```

```
REPEAT:
```

```
    addi t3, t3, 6      # t3 = t3 + 6  
    addi t3, t3, -1     # t3 = t3 - 1  
    andi t3, t3, 3      # t3 = t3 AND 3  
    beq zero, zero, REPEAT # Repeat the loop  
    nop
```

```
.end
```



8.4. Simulation of SW on RVfpgaNexys RTL model in Modelsim

ModelSim simulator allows users to simulate and debug hardware designs written in HDL (Hardware Description Language), such as VHDL and Verilog, providing detailed insights into the behavior of digital circuits.

ModelSim also allows hardware-software simulation. We will use ModelSim to observe program execution by tracking specific signals.

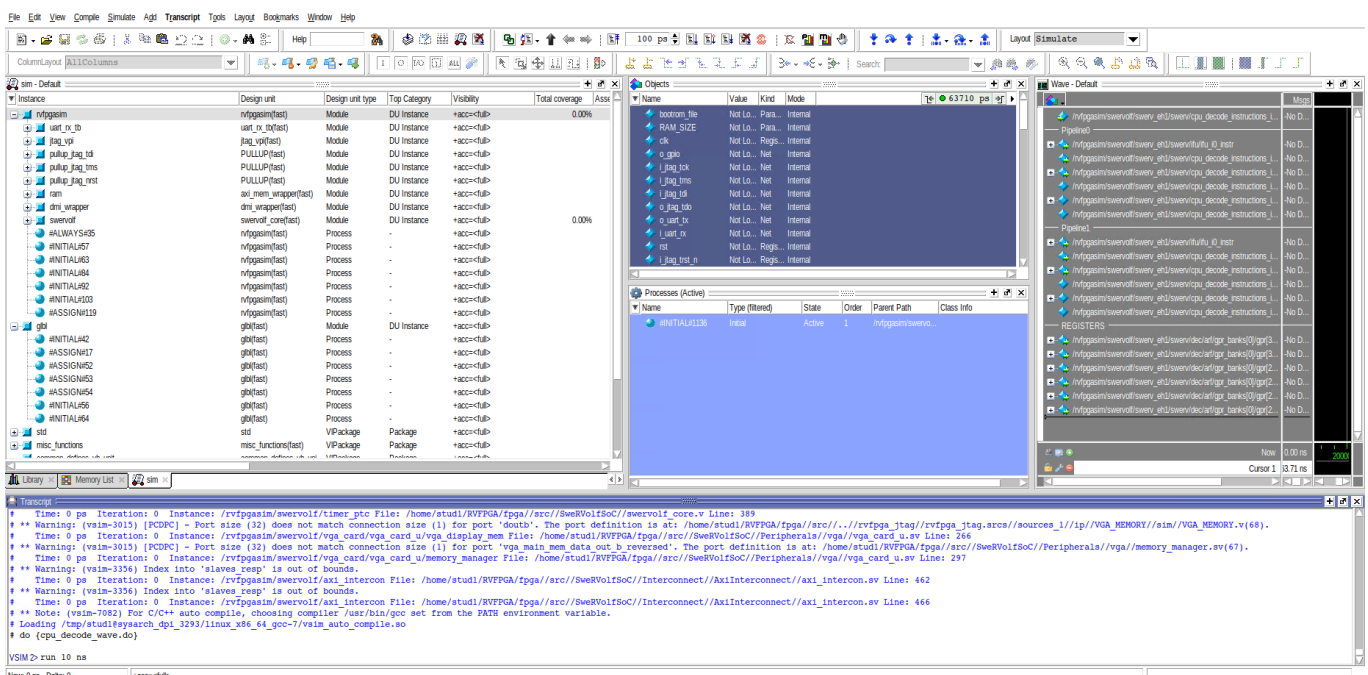
Please note that we will use the code image (binary file) of AL_Operations.S code from the previous section as the content of our code memory. Therefore, the simulation will only work if the build was completed successfully in the previous step.

To start the simulation:

- In Segger Select: **Build-> Rebuild AL_operations**
Note that you switched back to the original code of AL_Operations
- Run in the terminal:
 - a. `cd ~/RVFPGA/programs/meeting_1/AL_Operations/simulation`
 - b. `./rvfpgasim_simulate.sh AL_Operations &`



ModelSim will start running and automatically open the predefined signals.

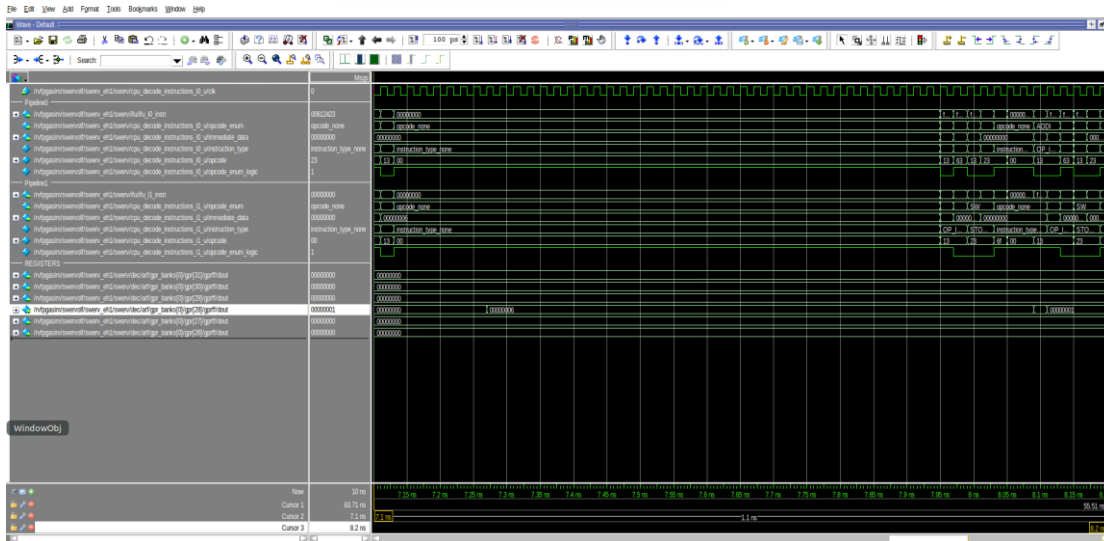
- In the Transcript window write : run 10 ns



- Use  to undock the wave window



- Zoom area 7.1 ns – 8.2 ns. Use   icons



Remember that the SWERV RISC-V core features a dual-pipeline architecture, which allows it to fetch program code simultaneously into both pipelines.

Register t3 is also called register 28 (x28)

```
li t3, 0x0      # t3 = 0 (00000e13) Pipeline0/rvfpgasim/swervolf/swerv_ah0/swerv/cpu_decode_instructions_i0_u/opcode_enum
REPEAT:
addi t3, t3, 6  # t3 = t3 + 6 (006e0e13) Pipeline1/rvfpgasim/swervolf/swerv_ah1/swerv/cpu_decode_instructions_i1_u/opcode_enum
addi t3, t3, -1 # t3 = t3 - 1 (fffe0e13) Pipeline0/rvfpgasim/swervolf/swerv_ah0/swerv/cpu_decode_instructions_i0_u/opcode_enum
andi t3, t3, 3  # t3 = t3 and 3 (003e7e13) Pipeline1/rvfpgasim/swervolf/swerv_ah1/swerv/cpu_decode_instructions_i1_u/opcode_enum
beq zero, zero, REPEAT # Repeat(fe000ae3) Pipeline0/rvfpgasim/swervolf/swerv_ah1/swerv/cpu_decode_instructions_i1_u/opcode_enum
nop
.end
```

Value of t3(x28) is in- /rvfpgasim/swervolf/swerv_ah1/swerv/dec/arf/gpr_banks[0]/gpr[28]/gprff/dout

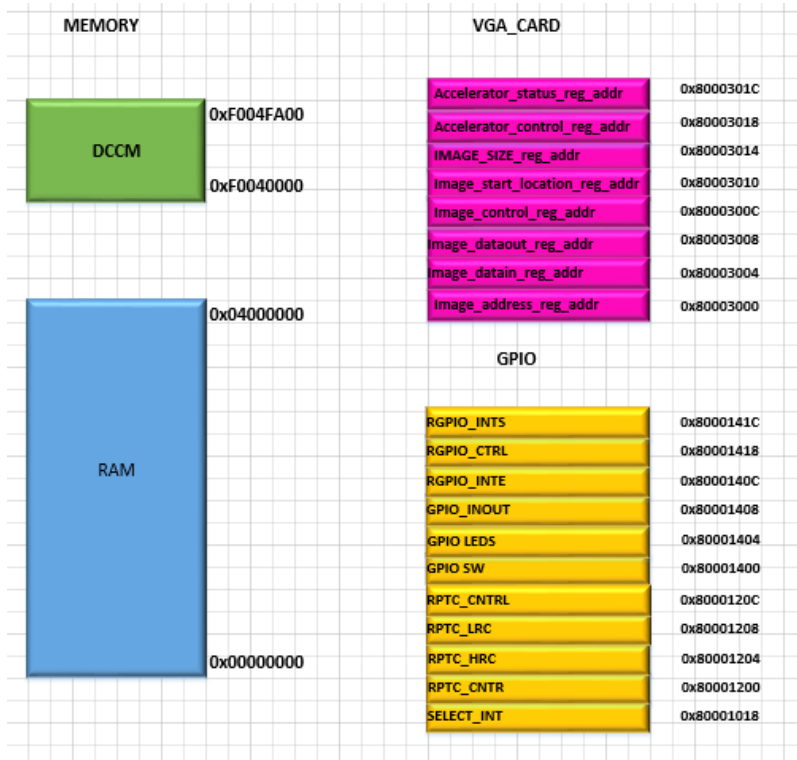
- Follow the program execution in the simulation and explain all the steps.
You can use this website - <https://www.bucknell.edu/~csci206/riscv-converter/> in order to convert hexadecimal code to RISC-V assembly instructions.

Answer Question #4

- Close Modelsim.



8.5. C programming

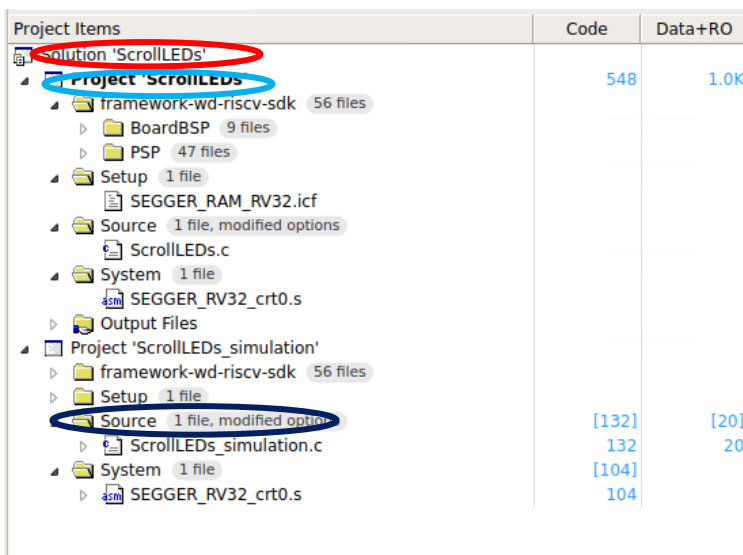


```
// memory-mapped I/O addresses
# GPIO_SWs = 0x80001400
# GPIO_LEDs = 0x80001404
# GPIO_INOUT = 0x80001408
//direction of GPIOs
```

SWERVOLF EH1 SOC MEMORY MAP

In SEGGER Embedded Studio:

Open the C program, **ScrollLEDs.c**, by clicking on **Open existing**, browse to **~/RVFPGA/programs/meeting_1/ScrollLEDs** and select **ScrollLEDs.emProject**





In **Project items** tab:

You can see **Solution 'ScrollLeds'** and 2 projects belonging there:




- Project **'ScrollLeds'**
- Project **'ScrollLeds_simulation'**

Project in **BOLD** is active


If 'Project 'ScrollLeds' isn't active

Select it->Mouse Right click-> Set as Active Project

Answer Question # 5

- Build the Program: **Build-> Rebuild ScrollLeds**
- Run the Program: **Debug->Go-** 
- When there is a yellow arrow  next to **int main (void)** Click on  **Continue Execution**

Answer Question # 6

- After finishing debug click on  to stop the program

Now we will run a simulation of the C project in ModelSim.

- To ensure the simulation run fast we need to change the delay settings. Select Project 'ScrollLeds_simulation' **->Mouse Right click-> Set as Active Project**
- The only difference with previous Project is the delay:

#define DELAY 0x30000

#define DELAY 0x30

- Build the Program: **Build-> Rebuilt ScrollLeds**
- In the terminal run
 - cd ~/RVFPGA/programs/meeting_1/ScrollLEDs/simulation
 - ./rvfpgasim_simulate.sh ScrollLeds_simulation &
- In the ModelSim transcript window: **run 1 us**

Answer Question # 7

- Close ModelSim.

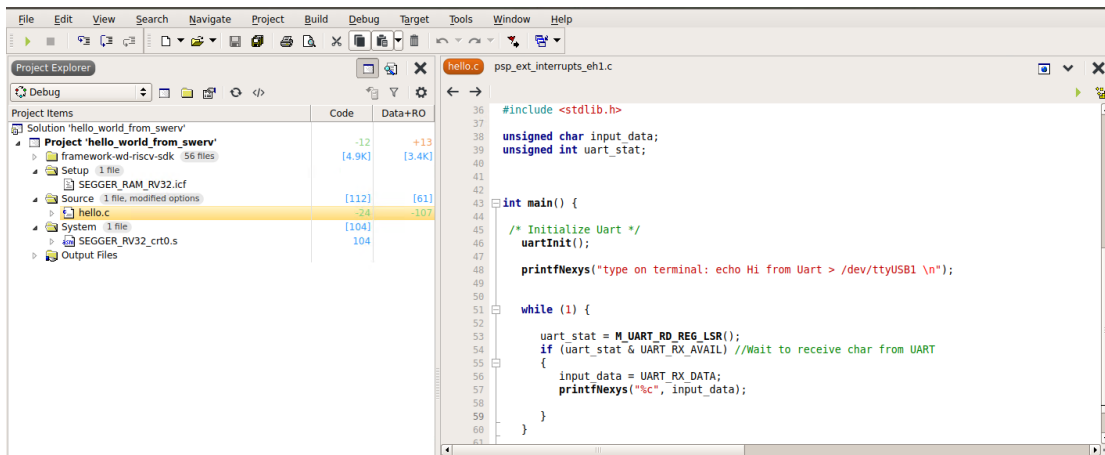


8.6. Hello world from Swerv C program

In this exercise, we will see how to use UART to debug the RISC-V code by sending and receiving data between the PC and Segger Studio Debugger.

On Segger ES:

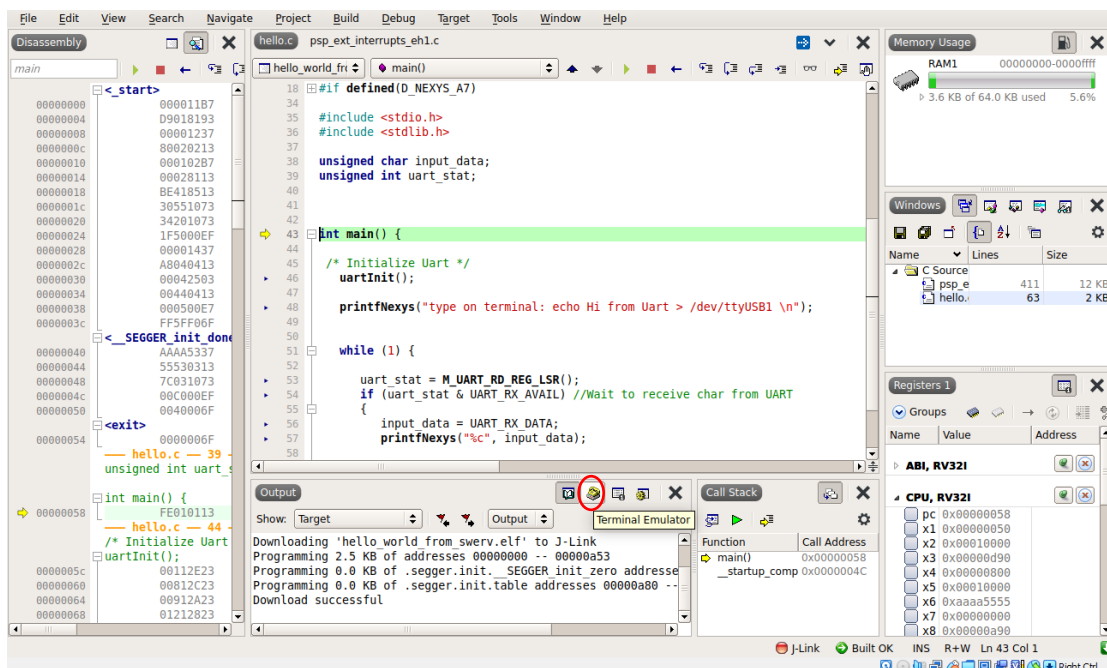
1. Open existing `~/RVFPGA/programs/meeting_1/hello_world_from_swerv`



2. Build-> Rebuild hellow_world_from_swervs

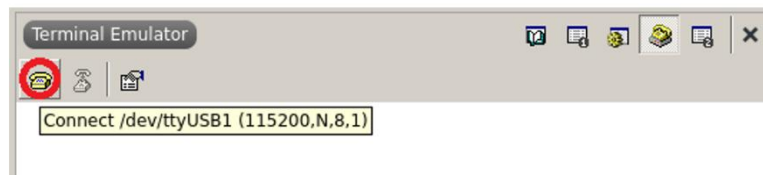
3. Debug -> Go


4. When there is a yellow arrow next to `int main()` Click on "Terminal Emulator" icon:





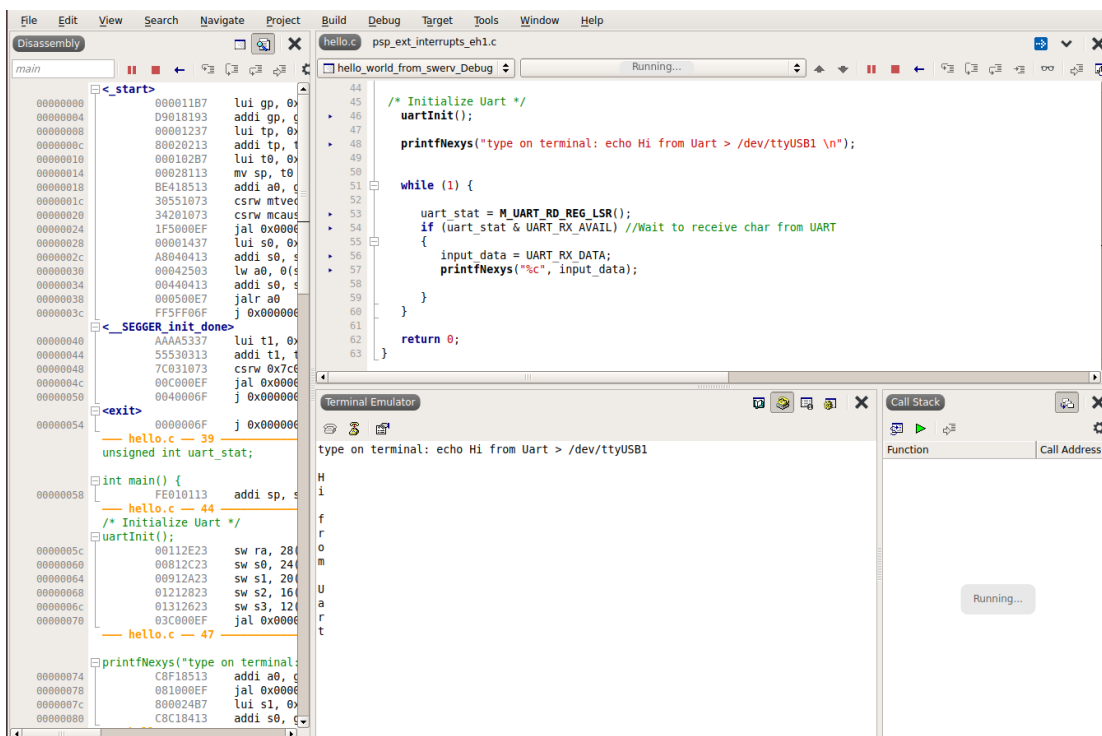
5. Click on left phone icon to connect the terminal:



6. Click  to run the program.

7. In UBUNTU terminal window type :

echo Hi from Uart > /dev/ttyUSB1



Modify the code so that it waits for a string of 16 characters and prints it on the terminal in one line.

Answer Question # 8



Exercises

Create your C programs by completing the following exercises.

Exercise 1. Howto access hardware devices Write a C program that flashes the value of the switches onto the LEDs. The value should pulse on and off slow enough that a person can view the flashing. Name the program **FlashSwitchesToLEDs.c**.

Exercise 2. 4 bit addition

Write a C program that displays the unsigned 4-bit addition of the 4 least significant bits of the switches and the 4 most significant bits of the switches. Display the result on the 4 least significant (right-most) bits of the LEDs. Name the program **4bitAdd.c**. The fifth bit of the LEDs should light up when unsigned overflow occurs (that is when the carry out is 1).