Faculty Of Engineering

The Hardware Design Laboratory

RVC_ASAP

RISC-V Core As-Soon-As-Possible

5-Stage Core Design implemented on FPGA

Gil Ya'akov

Matan Eshel

Fourth year project towards a bachelor's degree in Engineering

Advisor: Amichai Ben-David

Academic Advisor: Professor Adam Teman

October 2022

PREFACE - DOCUMENTATION STRUCTURE

The purpose of this document is to describe the RVC_ASAP project.
The project has many different aspects such as hardware, software, design, verification, tools, and flows.

The document main chapters:

- Overview & Motivation
- HAS | High-Level-Architecture-Specification (HW)
- MAS | Micro-Architecture-Specification (HW)
- Verification Plan (HW)
- FPGA
- SW & API
- TFM: Tools, Flow & Methodology

Gratitude's

First, we would like to thank our families, brothers and sisters who supported us all the way and expressed great interest in our work during the past year as part of the final project. In addition to this, we would like to thank our classmates who provided us with a listening ear and a source of new ideas. We will give the greatest thanks to the advisor of our final project, Amichai Ben-David. At the beginning of our journey in the project, we arrived with almost zero knowledge of writing and designing HW in the System Verilog language. Amichai invested a lot of time and effort to provide us with the knowledge and infrastructure on the subject so that we could develop the project independently. Amichai was available for any problem and any question at any time of the day and at unusual hours. In addition to this, Amichai spent a lot of time with us during the weekly meetings where we talked about the progress we made during the last week, problems that arose and directions for the way forward. We had the privilege to take part in this innovative, unconventional, and interesting project and we enjoyed working with a highly motivated advisor for the success of the project who is always there for us and constantly strives for excellence.

CONTENTS

Revision History

Rev. No.	Who	Description	Rev. Date
0.1	Gil Ya'akov & Matan Eshel	Initial RVC_ASAP project book.	09 September 2022
0.3	Gil Ya'akov	Fixes and completed the document structure.	11 September 2022
0.5	Gil Ya'akov	Completed MAS section and fixed rejects.	12 September 2022
0.7	Gil Ya'akov	Fixed rejects and completed VGA section.	13 September 2022
0.9	Gil Ya'akov	Added SW Guide chapter.	14 September 2022
1.1	Gil Ya'akov & Matan Eshel	Added FPGA chapter, read the book, edited proofreading, edited links, and fixed typos.	14 September 2022

Figures

Tables

Code snippet

Glossary

Term	Description
RVC_ASAP	RISC-V Core as soon as possible.
ISA	Instruction Set Architecture (such as X86, ARM, RISC-V).
IO	Input & output.
MMIO	Memory Mapped IO – method of performing Input/Output (IO) between the central processing unit (CPU) and peripheral devices in a computer.
IP	Intellectual Property. In this case, RTL building block that can be consumed.
HAS	High Level Architecture Specifications.
MAS	Micro Architecture Specifications.
I_MEM	Instruction Memory – where the program is loaded and ready for execution.
D_MEM	Data Memory – where the LOAD & STORE instructions read/write data.
CR_MEM	Control Register Memory – dedicated memory space for processor registers which changes or controls the general behavior of a CPU or other digital device.
VGA_MEM	Video Graphics Array Memory – dedicated memory space for communication with the screen.
Pipeline	Common way to parallel and utilize hardware. https://en.wikipedia.org/wiki/Instruction_pipelining
RISC	Reduce Instruction Set Computer (unlike CISC – Complex Instruction Set Computer). https://en.wikipedia.org/wiki/Reduced_instruction_set_computer
RISC-V	A relatively new open and free ISA (compared to intel X86, ARM). https://en.wikipedia.org/wiki/RISC-V
Unprivileged Spec	https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf
Privileged Spec	https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf
RV32I	RISC-V 32-bit Integer. The RISC-V baseline compatible ISA (no extensions M/A/F).
Standard interface	Functional characteristics to allow the exchange of information between two systems.
Word	32-bit of data – 4 bytes. The size of an integer in RV32I ISA.
Hazard	Potential source of harm. In the document when reading outdated data or wrongly executing instruction.
MSFF	Master-Slave Flip Flop.
API	Application Programming Interface.
TB	Test Bench.
SV	System Verilog Hardware Description Language.
ELF	Executable and Linkable Format.

Table 1 - Glossary

1 Overview & Motivation

1.1 Project Over-View

In this project we implemented a 5-stage processor that supports the RV32I ISA. The five stages of the processor are Fetch, Decode, Execute, Memory and Write Back. The processor architecture that we designed and implemented is essentially like the well-known MIPS architecture, although when we went down to details in the implementation of the hardware, we encountered architectural challenges for which we were required to produce creative solutions to ensure the correctness of RV32I ISA support. As a result, the lowest abstraction layer of the design (the level of the control signals and the interface with the peripheral components) received a unique character in relation to the MIPS architecture and other 5-stage architectures. The processor we designed contains a Hazard Detection unit and a Forwarding unit. In addition to this, the processor contains a 16kb instruction memory and a 16kb data memory divided into the following memory spaces: Data Memory, Control Register Memory, and Video Graphics Array (VGA) Memory. Due to the interface with the peripheral components found in the FPGA on which we burned the design, we implemented VGA and IO controllers.

To verify the correctness of the hardware implementation, we established a validation and simulation environment that operates fully through automation. That is, we wrote a program in the Bash language (script) that knows how to take an Assembly file or a C file and lead it through the entire chain of compilation and linking up to the level of producing two files in System Verilog format that make up the program's instruction memory and data memory. These files are loaded in the Test Bench, each to the memory space relevant to it, and after compiling the hardware, the program runs a simulation and checks the output that was produced in the memory in relation to the expected output. If the simulation was correct, the program indicates this, otherwise it will inform that an error occurred and indicate the location of the error. In this way, errors in the implementation of the hardware can be corrected.

In the advanced stages of the project, after the stabilization of the architecture, we synthesized and burned the design on an Intel DE10-Lite FPGA and built Hardware API libraries for communication through software with the hardware. In addition to this, we built a game that runs on top of the processor. In designing the game, we used ASCII tables and symbols that we created and loaded into the program's memory. For this purpose, we used game design tools that were used in the past to design arcade games. The main challenge at this stage of the project was designing a computer game in a system with few resources (in terms of the memory and peripheral components available to us) and we were required to produce effective solutions in the software code to carry out the task.

The entire RTL code of the project, documentations, applications, validation and verification tools, and guides to operate them are all can be founded in the RVC_ASAP GitHub repository.

1.2 MOTIVATION

RISC-V ISA

Due to the growing importance of RISC-V ISA we conceived a project where we could study the RISC-V ISA and reveal how simple and accessible the ability to design your CPU is. In the initial stage of the project, we were required to implement in hardware the data flow of each instruction in the ISA. As a result, we studied each instruction in the ISA deeply and learned the meaning of each instruction. Building a RISC-V based processor is a project accepted and known to many undergraduate students around the world. What sets this project apart is that we built everything ourselves and adhered to advanced methods of hardware planning and a readable coding style.

Git

The work methodology with Git nowadays is very common. As a Computer Engineers, most of our work is done with Git. Given that, one of our goals was to improve our ability in this environment. Our final project was fully managed with Git tool, mainly for the purposes of smart and efficient management and creating complete uniformity among the team members. Working with Git as part of the project contributed a lot to our ability to work with this tool.

System-Verilog HDL

One of the most important skills to acquire after completing a degree in Computer or Electrical Engineering is the ability to design and implement hardware. The purpose of this ability is to implement new ideas that come to the mind of the engineer by implementing them in hardware. The System-Verilog language is a hardware description language that allows you to implement the hardware just as you write a computer program. With this understanding, our goal was to acquire knowledge in the System-Verilog language and improve our ability to implement ideas that require a hardware implementation. The processor we built was written in the System-Verilog language and as a result we learned a lot about writing hardware in this language.

HW-SW Integration

An important point to understand in computer architecture is the interface between the hardware and the software. The ability to write an efficient and optimal program in terms of runtime complexity and memory complexity depends directly on knowing the hardware architecture on which the program code executes and knowing the limitations of the architecture. A central motivation with which we approached the project was to improve our understanding in this area. To check the correctness of the hardware implementation, we wrote test programs in Assembly and C languages. To write the test programs, we were required to know all the Assembly instructions relevant to our architecture and to understand how compiler and assembler works. In addition to this, we were required to understand how to write a linker script and, in general, to understand how the compilation and linking chain works. At an advanced stage of the project, we implemented Hardware API libraries for communication through the software with the hardware and wrote a computer game so that we were required to develop creative solutions to consider the limitations of the architecture.

2 HAS | HIGH-LEVEL-ARCHITECTURE-SPECIFICATION (HW)

2.1 HIGH LEVEL DEFINITION

In this High-Level Architecture Specification (HAS) section we will describe the specification of RVC_ASAP IP – a RISC-V Core supported RV32I ISA. The section will explain the macro design of the core, the use and functionality of the different components in it.

This section is architectonic section, and it is a macro view section of the project. To fully understand the micro level architecture, please read the MAS | Micro-Architecture-Specification (HW) as well.

2.2 GENERAL BACKGROUND

2.2.1 RISC-V & ISA background

RISC-V is an open-source base integer instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. It is a classic RISC architecture rebuilt for modern times. At its heart is an array of 32 registers containing the processor's running state, the data is immediately operated on, and housekeeping information. RISC-V comes in 32-bit and 64-bit variants, with register size to match. The project began in 2010 at the University of California, Berkeley along with many volunteer contributors not affiliated with the university. It was originally designed to support computer architecture research and education but eventually on nowadays used for industry and many other uses. The RISC-V eco system has all the SW needed to program, compile and creating RISC-V assembly & executable RISC-V machine code. Unlike outer ISAs, anyone can write compatible RISC-V core without going through bureaucracy of licenses and fees. RVC_ASAP supports the RV32I ISA that can be found in this link – RISC-V spec. RISC-V is a load-store architecture, meaning three things:

1. Its arithmetic instructions operate only on registers.
2. Only load and store instructions transfer data to and from memory.
3. Data must first be loaded into a register before it can be operated on.

RV32I ISA is shown in Figure 1.

2.2.2 Central Processing Unit (CPU)

General-Purpose CPU is the electronic circuitry that executes instructions comprising a computer program. The CPU performs basic arithmetic, logic, controlling, and input/output (IO) operations specified by the instructions in the program. Principal components of a CPU include the arithmetic–logic unit (ALU) that performs arithmetic and logic operations, processor registers that supply operands to the ALU and store the results of ALU operations, and a control unit that orchestrates the fetching (from memory), decoding and execution of instructions by directing the coordinated operations of the ALU, registers and other components.

RV32I Base Instruction Set

imm[31:12]			rd	0110111	LUI
imm[31:12]			rd	0010111	AUIPC
imm[20|10:1|11|19:12]			rd	1101111	JAL
imm[11:0]	rs1	000	rd	1100111	JALR
imm[12|10:5] rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5] rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5] rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5] rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5] rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5] rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU
imm[11:5] rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5] rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5] rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI
0000000 shamt	rs1	001	rd	0010011	SLLI
0000000 shamt	rs1	101	rd	0010011	SRLI
0100000 shamt	rs1	101	rd	0010011	SRAI
0000000 rs2	rs1	000	rd	0110011	ADD
0100000 rs2	rs1	000	rd	0110011	SUB
0000000 rs2	rs1	001	rd	0110011	SLL
0000000 rs2	rs1	010	rd	0110011	SLT
0000000 rs2	rs1	011	rd	0110011	SLTU
0000000 rs2	rs1	100	rd	0110011	XOR
0000000 rs2	rs1	101	rd	0110011	SRL
0100000 rs2	rs1	101	rd	0110011	SRA
0000000 rs2	rs1	110	rd	0110011	OR
0000000 rs2	rs1	111	rd	0110011	AND
fm pred succ	rs1	000	rd	0001111	FENCE
000000000000	00000	000	00000	1110011	ECALL
000000000001	00000	000	00000	1110011	EBREAK

Figure 1 - RV32I ISA

2.3 High Level Description

RVC_ASAP, which all this project converse to, is our own implementation of a general-purpose compute unit based on RV32I ISA which supports communication with FPGA peripherals, screen, and memory modules.

- RVC_ASAP include 2 Main building blocks: Core and Memory Wrapper.
- The entire module and building blocks architecture based on synchronic pipeline architecture.
- The pipeline is in-order pipeline (instruction performs by their order in program).
- The pipeline is single scalar, only one instruction is executed per clock cycle.
- Core: 5-Stage pipeline RV32I core. The core contains hazard detection and forwarding units.
- Memory Wrapper contains:
    - Instruction Memory which the core reads the 32-bit RV32I instructions from.
    - Data Memory which the core use to initiate LOAD & STORE instruction.
    - Control Registers Memory which handles the communication with the FPGA peripherals.
    - Video Graphic Array Controller which handles the communication with the screen and contains VGA memory.
- The design is compatible with RV32I ISA.
- Steady State Instructions Per Cycle (IPC) of 1 (probably ~0.7 IPC).
- The access to the memory is word-aligned (4 bytes).

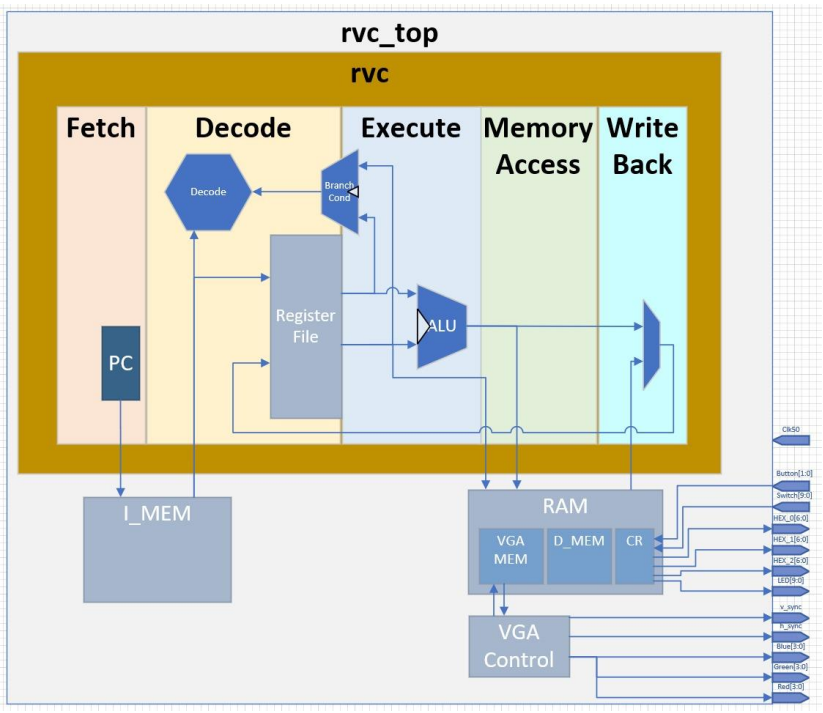

Figure 2 - Top level block diagram

2.4 Design Building Blocks

2.4.1 Core & pipeline stages

The core is the building block that runs and executes the program instructions. It is RV32I based core, which means the core entire design supports only RV32I instructions – 32-bit encoded assembly instructions. The core contains hazard detection and forwarding units.

RVC_ASAP core is a piped core with 5 stages:

1. Instruction Fetch (Q100H)
2. Decode (Q101H)
3. Execute (Q102H)
4. Memory (Q103H)
5. Write Back (Q104H)

2.4.2 Memory regions

RVC_ASAP has total of 69.5kb S-RAM memory in addition to Flip-Flop memories and Registers File. The entire memory is divided into 4 regions:

1. I_MEM – 16kb.
2. D_MEM – 16kb.
3. CR_MEM – as part of Data Memory allocation.
4. VGA_MEM – 32.5kb.

Each memory transaction has a 32-bit address. Please see Memory regions in the MAS to understand how we assigned the CPU address to the appropriate memory space.

Figure 3 shows us the distribution of the memory space as it is depicted from the processor's point of view.

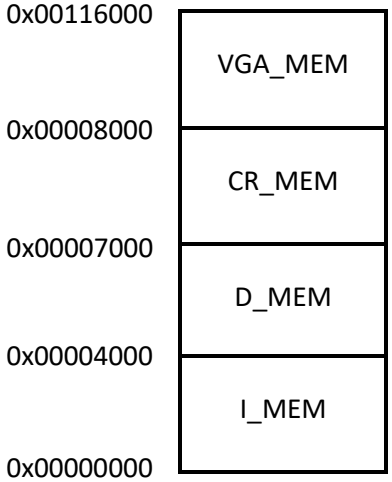

Figure 3 - Address space from core's point of view

Table 2 shows us the exact memory regions ranges from the core's point of view.

Region	Start address	End address	Description
I_MEM	0x00000000	0x00003FFF	16kb of Instruction Memory.
D_MEM	0x00004000	0x00006FFF	16kb of Data Memory.
CR_MEM	0x00007000	0x00007FFF	Control Register Memory as part of Data Memory allocation.
VGA_MEM	0x00008000	0x000115FF	32.5kb of VGA Memory.

Table 2 - Memory regions range from core's point of view

2.4.3 VGA Controller

The VGA controller generates synchronization signals and reads the VGA Memory. The VGA Controller contains the following instances:

- sync_gen.
- Pll_2.
- VGA_MEM.

For more information, please read the VGA Controller section in the MAS.

3 MAS | MICRO-ARCHITECTURE-SPECIFICATION (HW)

3.1 MICRO LEVEL DEFINITION

The MAS describes the Micro-Level Architecture of the RVC_ASAP. This is the logic design implementation of RVC_ASAP architectonic demands described on HAS. This section covers the micro architecture of the core's stages, and the way that mem_wrap interact with the core. The section will cover the logic design for this communication and the implementation of the RVC_ASAP.

3.2 MICRO LEVEL DESCRIPTION

RVC_ASAP include 2 main building blocks: Core and Memory Wrapper. Those building blocks wrapped together in the top view of the design. The Memory Wrapper wraps together the Instruction Memory, Data Memory, CR Memory, and VGA Controller. The interface between the core and the Memory Wrapper located in the top view of the design. The core is 5-stage piped core which means that the interface between the core and the Memory Wrapper is expressed in 4 stages. The first stage is the Fetch stage. The core sends the address of the next instruction which pointed by the PC (Program Counter) to the Memory Wrapper. The memory concludes that this is an address intended for the Instruction Memory area and transmits the signal to this memory. The Instruction Memory module will provide the instruction to the core at the next clock cycle. The second stage of the interface between the core and the Memory Wrapper is in the Decode stage, where the core gets the instruction from the Instruction Memory. The next stage is the Memory stage. In this stage the core may want to perform any memory operations such as store data in the memory or load data from a memory address. The requests for memory operations go through the Memory Wrapper which checks the target address for the operation and sends the signals only to the appropriate memory according to the distribution of the address space (Data Memory, CR Memory, or VGA Memory). The final stage is the Write Back stage where if we loaded a data from the memory in the Memory stage, then the data arrives in the Write Back stage.

The top view of the design outputs the control signals for communication with the FPGA so that some of them are input signals and some are output signals.

The flow that has been described so far is the main flow of the design. Now, we will enter the implementation level of the various design components and explain the interface between them.

3.3 Design Building Blocks

3.3.1 Parameters

RVC_ASAP contains the following parameters which are defined in the rvc_asap_pkg file:

Parameter	Value	Description
I_MEM_MSB	0x00003FFF	Instruction Memory section highest address.
D_MEM_MSB	0x00006FFF	Data Memory section highest address.
CR_MEM_MSB	0x00007FFF	CR Memory section highest address.
VGA_MEM_MSB	0x000115FF	VGA Memory section highest address.
LSB_REGION	0	Least Significant Bit of the memory addresses.
MSB_REGION	15	Most Significant Bit of the memory addresses.
VGA_MSB_REGION	19	Most Significant Bit of the VGA Memory address.
I_MEM_REGION_FLOOR	0	Instruction Memory section lowest address.
I_MEM_REGION_ROOF	0x00003FFF	Instruction Memory section highest address.
D_MEM_REGION_FLOOR	0x00004000	Data Memory section lowest address.
D_MEM_REGION_ROOF	0x00006FFF	Data Memory section highest address.
CR_MEM_REGION_FLOOR	0x00007000	CR Memory section lowest address.
CR _MEM_REGION_ROOF	0x00007FFF	CR Memory section highest address.
VGA_MEM_REGION_FLOOR	0x00008000	VGA Memory section lowest address.
VGA _MEM_REGION_ROOF	0x000115FF	VGA Memory section highest address.
SIZE_D_MEM	3000h	Data Memory size in bytes.
SIZE_VGA_MEM	38400d	VGA Memory size in bytes.
CR_SEG7_0	0x00007000	CR address, RW 7 bit.
CR_SEG7_1	0x00007004	CR address, RW 7 bit.
CR_SEG7_2	0x00007008	CR address, RW 7 bit.
CR_SEG7_3	0x0000700C	CR address, RW 7 bit.
CR_SEG7_4	0x00007010	CR address, RW 7 bit.
CR_SEG7_5	0x00007014	CR address, RW 7 bit.
CR_LED	0x00007018	CR address, RW 10 bit.
CR_Button_0	0x0000701C	CR address, RO 1 bit.
CR_Button_1	0x00007020	CR address, RO 1 bit.
CR_Switch	0x00007024	CR address, RO 10 bit.
CR_CURSOR_H	0x00007028	CR address, RW 32 bit.
CR_CURSOR_V	0x0000702C	CR address, RW 32 bit.

Table 3 - RVC_ASAP parameters

These parameters were defined for the flexibility of the design regarding the changes and readability of the code.

3.3.2 Top level

The top level of the design contains the interface and the communication between the FPGA and the RVC_ASAP component. The Figure below illustrates this communication:

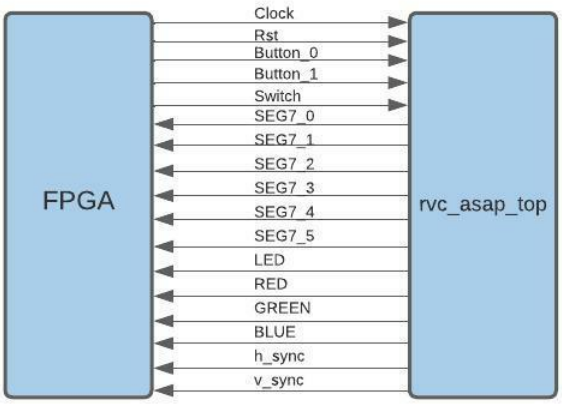

Figure 4 - Top Level interface

Signal	Size [bits]	Description
Clock	1	Clock signal.
Rst	1	Reset signal.
Button_0	1	Key 0.
Button_0	1	Key 1.
Switch	10	The 10 switch buttons.
SEG7_0	8	The 8 bulbs of segment 0.
SEG7_1	8	The 8 bulbs of segment 1.
SEG7_2	8	The 8 bulbs of segment 2.
SEG7_3	8	The 8 bulbs of segment 3.
SEG7_4	8	The 8 bulbs of segment 4.
SEG7_5	8	The 8 bulbs of segment 5.
LED	10	The 10 led bulbs.
RED	4	The red signal of the VGA.
GREEN	4	The green signal of the VGA.
BLUE	4	The blue signal of the VGA.
h_sync	1	Horizontal synchronization signal of the VGA.
V_sync	1	Vertical synchronization signal of the VGA.

Table 4 - rvc_asap_top - FPGA interface signals

3.3.3 Core – Memory Wrapper interface

As mentioned earlier, the Memory Wrapper wraps together the Instruction Memory, Data Memory, CR Memory, and VGA Controller. The interface between the core and the Memory Wrapper located in the top view of the design. All the information traffic between the core and the various memories goes through the Memory Wrapper. The Figure below illustrates this interface:

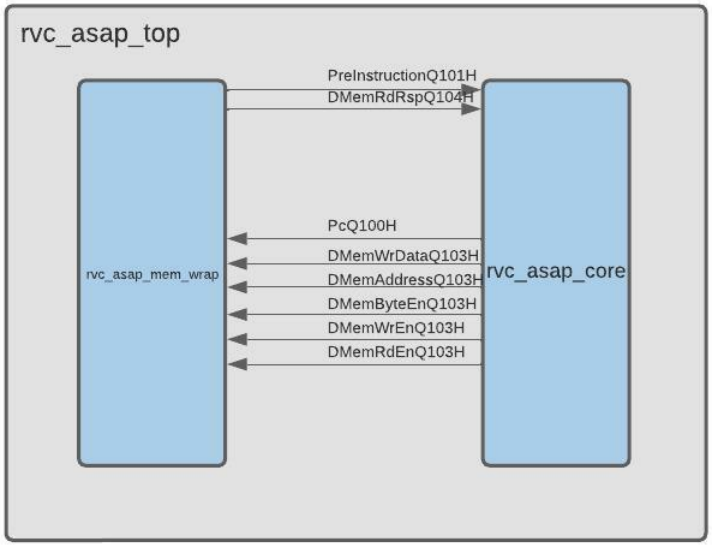

Figure 5 - Core - Memory Wrapper interface

Signal	Size [bits]	Description
PreInstructionQ101H	32	Pre-instruction to decode.
DMemRdRspQ104H	32	Data to read from the memory.
PcQ100H	32	Program Counter.
DMemWrDataQ103H	32	Data to write to the memory.
DMemAddressQ103H	32	Address.
DMemByteEnQ103H	4	Byte enable – defines which bytes to choose from the 4 bytes.
DMemWrEnQ103H	1	Write enable.
DMemRdEnQ103H	1	Read enable.

Table 5 - rvc_asap_core - rvc_asap_mem_wrap interface signals

3.3.4 Core & pipeline stages

The core is one of the main building blocks of RVC_ASAP. The core task is to run and execute the program instructions. The core is in order pipeline architecture with 5 stages. The MAS covers the core stages along with the logic and interface with the Memory Wrapper. The interface of the core, in addition to what is shown in Figure 5, also receives a clock signal and a reset signal.

3.3.4.1 Instruction Fetch (Q100H)

- Send PC (Program Counter) to Instruction Memory.
- Set Next PC – (PC + 4) or calculated address.

```
assign Pc_To_ImemQ100H  = PcQ100H;
assign PcPlus4Q100H     = PcQ100H + 3'h4;
`RVC_EN_RST_MSFF(PcQ100H, NextPcQ102H, Clock, PcEnQ101H, Rst)
```

PcQ100H is the actual PC that sent to Instruction Memory. The branch resolution is in Execute stage (Q102H), so there are two options. If the branch condition met – in the next clock cycle PcQ100H will get the jump target address, and if it not met – PcQ100H will get PcQ100H + 4.

3.3.4.2 *Decode (Q101H)*

- Load & Control hazard detection:

```
assign PreRegSrc1Q101H           = PreInstructionQ101H[19:15];
assign PreRegSrc2Q101H           = PreInstructionQ101H[24:20];
assign LoadHzrdDetectQ101H       = Rst ? 1'b0 :
                                   ((PreRegSrc1Q101H == RegDstQ102H) && (OpcodeQ102H == LOAD)) ? 1'b1:
                                   ((PreRegSrc2Q101H == RegDstQ102H) && (OpcodeQ102H == LOAD)) ? 1'b1:
                                                                                                1'b0;

assign PcEnQ101H                 = !LoadHzrdDetectQ101H;
assign InstructionQ101H          = flushQ102H ? NOP :
                                   flushQ103H ? NOP :
                                   LoadHzrdDetectQ101H ? NOP:
                                   LoadHzrdDetectQ102H ? PreviousInstructionQ101H :
                                                         PreInstructionQ101H;
```

We get the PreInstructionQ101H from the Instruction Memory. We look on the RegSrc1 and the RegSrc2 fields of the instruction. LoadHzrdDetectQ101H checks first if we are in reset, if yes there is no meaning to check for load hazard, hence it gets the value of 0. Otherwise, it checks if the instruction in the Execute stage is a LOAD instruction and it writes to a register RegDst which is match to RegSrc1 or RegSrc2, if yes, we are in load hazard, otherwise not.
PcEnQ101H checks if there is no load hazard – if there is no load hazard allow the Q100H stage registers to sample on the rise of the clock. If we find out in Execute stage that the branch condition is met, we will need to insert 2 NOP (No Operation) instructions, one is to don't decode the current instruction and the second is to don't decode the current fetched instruction. In other words, we have a branch predictor where his jump rule is always not taken. If we are not in one of the flushes, we check if we are in load hazard. If yes, we insert one NOP and if not, we are check if we were in load hazard in the previous step, if yes, we will complete the instruction that was waiting – PreviousInstructionQ101H. Otherwise, proceed regular and take the PreInstructionQ101H.

- Receiving instruction from Instruction Memory and using decoder to turn on the relevant control bits:

```
assign OpcodeQ101H           = t_opcode'(InstructionQ101H[6:0]);
assign Funct3Q101H           = InstructionQ101H[14:12];
assign Funct7Q101H           = InstructionQ101H[31:25];
assign SelNextPcAluOutJQ101H = (OpcodeQ101H == JAL) || (OpcodeQ101H == JALR);
assign SelNextPcAluOutBQ101H = (OpcodeQ101H == BRANCH);
assign SelRegWrPcQ101H       = (OpcodeQ101H == JAL) || (OpcodeQ101H == JALR);
assign SelAluPcQ101H         = (OpcodeQ101H == JAL) || (OpcodeQ101H == BRANCH)
                                                    || (OpcodeQ101H == AUIPC);
assign SelAluImmQ101H        =!(OpcodeQ101H == R_OP); // Only in case of RegReg Operation the Imm
Selector is deasserted - defualt is asserted
assign SelDMemWbQ101H        = (OpcodeQ101H == LOAD);
assign CtrlLuiQ101H          = (OpcodeQ101H == LUI);
assign CtrlRegWrEnQ101H      = (OpcodeQ101H == LUI ) || (OpcodeQ101H == AUIPC) ||
                               (OpcodeQ101H == JAL)  || (OpcodeQ101H == JALR) ||
                               (OpcodeQ101H == LOAD) || (OpcodeQ101H == I_OP)  ||
                               (OpcodeQ101H == R_OP) || (OpcodeQ101H == FENCE);
assign CtrlDMemWrEnQ101H     = (OpcodeQ101H == STORE);
assign CtrlSignExtQ101H      = (OpcodeQ101H == LOAD) && (!Funct3Q101H[2]); // Sign extend the LOAD
from memory read.
assign CtrlDMemByteEnQ101H   = ((OpcodeQ101H == LOAD) || (OpcodeQ101H == STORE)) &&
                               (Funct3Q101H[1:0] == 2'b00) ? 4'b0001 : // LB || SB
                               ((OpcodeQ101H == LOAD) || (OpcodeQ101H == STORE)) &&
                               (Funct3Q101H[1:0] == 2'b01) ? 4'b0011 : // LH || SH
                               ((OpcodeQ101H == LOAD) || (OpcodeQ101H == STORE)) &&
                               (Funct3Q101H[1:0] == 2'b10) ? 4'b1111 : // LW || SW
                                                             4'b0000 ;
assign CtrlBranchOpQ101H     = t_branch_type'(Funct3Q101H);
```

In this section we fill in the various control singles according to the instruction that was received and according to the fields as they are defined in ISA for each group of commands with a similar structure. Using these control signals, we activate the various components.

- Config the ALU operation in case of R_TYPE instruction:

```
always_comb begin
    unique casez ({Funct3Q101H, Funct7Q101H, OpcodeQ101H})
    // ---- R type ----
    {3'b000, 7'b0000000, R_OP} : CtrlAluOpQ101H = ADD;  // ADD
    {3'b000, 7'b0100000, R_OP} : CtrlAluOpQ101H = SUB;  // SUB
    {3'b001, 7'b0000000, R_OP} : CtrlAluOpQ101H = SLL;  // SLL
    {3'b010, 7'b0000000, R_OP} : CtrlAluOpQ101H = SLT;  // SLT
    {3'b011, 7'b0000000, R_OP} : CtrlAluOpQ101H = SLTU; // SLTU
    {3'b100, 7'b0000000, R_OP} : CtrlAluOpQ101H = XOR;  // XOR
    {3'b101, 7'b0000000, R_OP} : CtrlAluOpQ101H = SRL;  // SRL
    {3'b101, 7'b0100000, R_OP} : CtrlAluOpQ101H = SRA;  // SRA
    {3'b110, 7'b0000000, R_OP} : CtrlAluOpQ101H = OR;   // OR
    {3'b111, 7'b0000000, R_OP} : CtrlAluOpQ101H = AND;  // AND
    // ---- I type ----
    {3'b000, 7'b???????, I_OP} : CtrlAluOpQ101H = ADD;  // ADDI
    {3'b010, 7'b???????, I_OP} : CtrlAluOpQ101H = SLT;  // SLTI
    {3'b011, 7'b???????, I_OP} : CtrlAluOpQ101H = SLTU; // SLTUI
    {3'b100, 7'b???????, I_OP} : CtrlAluOpQ101H = XOR;  // XORI
    {3'b110, 7'b???????, I_OP} : CtrlAluOpQ101H = OR;   // ORI
    {3'b111, 7'b???????, I_OP} : CtrlAluOpQ101H = AND;  // ANDI
    {3'b001, 7'b0000000, I_OP} : CtrlAluOpQ101H = SLL;  // SLLI
    {3'b101, 7'b0000000, I_OP} : CtrlAluOpQ101H = SRL;  // SRLI
    {3'b101, 7'b0100000, I_OP} : CtrlAluOpQ101H = SRA;  // SRAI
    // ---- Other ----
    default                    : CtrlAluOpQ101H = ADD;  // LUI || AUIPC || JAL || JALR || BRANCH || LOAD || STORE
    endcase
end
```

- Building the immediate types:

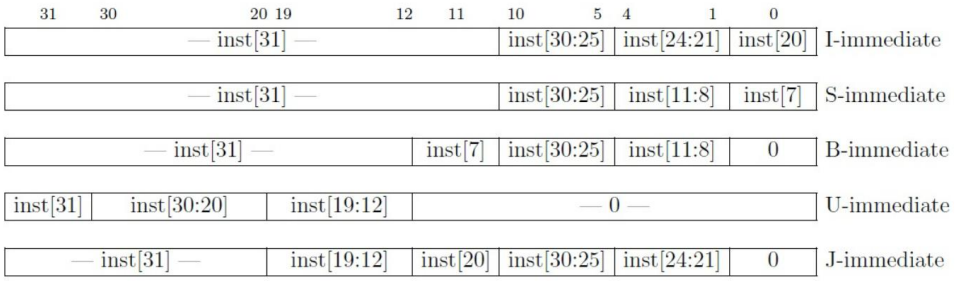

Figure 6 - Immediate types

Figure 6: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

```
always_comb begin
  unique casez (OpcodeQ101H) // Mux
    JALR, I_OP, LOAD : SelImmTypeQ101H = I_TYPE;
    LUI, AUIPC       : SelImmTypeQ101H = U_TYPE;
    JAL              : SelImmTypeQ101H = J_TYPE;
    BRANCH           : SelImmTypeQ101H = B_TYPE;
    STORE            : SelImmTypeQ101H = S_TYPE;
    default          : SelImmTypeQ101H = I_TYPE;
  endcase
  unique casez (SelImmTypeQ101H) // Mux
    U_TYPE : ImmediateQ101H = {    InstructionQ101H[31:12], 12'b0 } ;
    I_TYPE : ImmediateQ101H = {{20{InstructionQ101H[31]}}, InstructionQ101H[31:20]};
    S_TYPE : ImmediateQ101H = {{20{InstructionQ101H[31]}}, InstructionQ101H[31:25],InstructionQ101H[11:7]  };
    B_TYPE : ImmediateQ101H = {{20{InstructionQ101H[31]}}, InstructionQ101H[7]    ,InstructionQ101H[30:25], InstructionQ101H[11:8], 1'b0};
    J_TYPE : ImmediateQ101H = {{12{InstructionQ101H[31]}}, InstructionQ101H[19:12],InstructionQ101H[20]    , InstructionQ101H[30:21, 1'b0};
    default: ImmediateQ101H = {   InstructionQ101H[31:12], 12'b0 }; // U_Immediate
  endcase
```

- Using RS1 and RS2 to read the Register File:

```
assign RegDstQ101H  = InstructionQ101H[11:7];
assign RegSrc1Q101H = InstructionQ101H[19:15];
assign RegSrc2Q101H = InstructionQ101H[24:20];
// ---- Read Register File ----
assign MatchRd1AftrWrQ101H = (RegSrc1Q101H == RegDstQ104H) && (CtrlRegWrEnQ104H) && (RegSrc1Q101H != 5'b0);
assign RegRdData1Q101H     = MatchRd1AftrWrQ101H     ? RegWrDataQ104H       : // forwarding
                             (RegSrc1Q101H == 5'b0) ? 32'b0                :
                                                      Register[RegSrc1Q101H];
assign MatchRd2AftrWrQ101H = (RegSrc2Q101H == RegDstQ104H) && (CtrlRegWrEnQ104H) && (RegSrc2Q101H != 5'b0);
assign RegRdData2Q101H     =  MatchRd2AftrWrQ101H    ? RegWrDataQ104H       : // forwarding
                            (RegSrc2Q101H == 5'b0) ? 32'b0                :
                                                     Register[RegSrc2Q101H];
```

Here we are also making forwarding from Write Back stage (Q104H) directly to RegRdData1Q101H and RegRdData2Q101H signals which read the Register File.

3.3.4.3 Execute (Q102H)

- Hazard Detection:

```
assign Hazard1Data1Q102H = (RegSrc1Q102H == RegDstQ103H) && (CtrlRegWrEnQ103H) && (RegSrc1Q102H != 5'b0);
assign Hazard2Data1Q102H = (RegSrc1Q102H == RegDstQ104H) && (CtrlRegWrEnQ104H) && (RegSrc1Q102H != 5'b0);
assign Hazard1Data2Q102H = (RegSrc2Q102H == RegDstQ103H) && (CtrlRegWrEnQ103H) && (RegSrc2Q102H != 5'b0);
assign Hazard2Data2Q102H = (RegSrc2Q102H == RegDstQ104H) && (CtrlRegWrEnQ104H) && (RegSrc2Q102H != 5'b0);
```

Here we check if one of the operands RegSrc1Q102H or RegSrc2Q102H updated value is going to be updated by one of the instructions from the Memory stage or from the Write Back stage, if yes, we are in data hazard. We also check that the RegSrc is not the x0 registers because x0 register must consist of the 0 value and we don't want to make nonzero value forwarding.

- Forwarding Unit:

```
assign RegRdData1Q102H = Hazard1Data1Q102H ? AluOutQ103H       : // Rd 102 After Wr 103
                         Hazard2Data1Q102H ? RegWrDataQ104H    : // Rd 102 After Wr 104
                                             PreRegRdData1Q102H; // Common Case - No Hazard
assign RegRdData2Q102H = Hazard1Data2Q102H ? AluOutQ103H       : // Rd 102 After Wr 103
                         Hazard2Data2Q102H ? RegWrDataQ104H    : // Rd 102 After Wr 104
                                             PreRegRdData2Q102H; // Common Case - No Hazard
```

Here we define mux for each one of the signals RegRdData1Q102H and RegRdData2Q102H. The mux checks first if we are in Hazard1Data which means we read register that going to be updated by instruction which located in the Memory stage, if yes, forward the AluOut103H value. Otherwise, checks if we are in Hazard2Data which means we read register that going to be updated by instruction which located in the Write Back stage, if yes, forward the RegWrDataQ104H value. Note that a situation where we read a register in the Execute stage is not possible so that there is a LOAD instruction in the Memory stage which writes to this register and apparently should be forwarding although this is not possible because the sampling is done only at the end of the current clock cycle. This is not possible since this is a load hazard which is handled by spacing between the LOAD instruction and the next instruction that needs the LOAD value by NOP instruction, therefore the forwarding situation from the Memory stage to the Execute stage is necessarily of instructions that do not access memory.

- ALU operations: Calculation of the information for writing back to the Register File, address calculation for Store/Load instructions, calculation of target address for Jump/Branch:

```
assign AluIn1Q102H = SelAluPcQ102H  ? PcQ102H          : RegRdData1Q102H;
assign AluIn2Q102H = SelAluImmQ102H ? ImmediateQ102H   : RegRdData2Q102H;

always_comb begin : alu_logic
  ShamtQ102H      = AluIn2Q102H[4:0];
  unique casez (CtrlAluOpQ102H)
    // Adder
    ADD     : AluOutQ102H = AluIn1Q102H +   AluIn2Q102H;                          //
ADD/LW/SW/AUIOC/JAL/JALR/BRANCH/
    SUB     : AluOutQ102H = AluIn1Q102H + (~AluIn2Q102H) + 1'b1;                  // SUB
    SLT     : AluOutQ102H = {31'b0, ($signed(AluIn1Q102H) < $signed(AluIn2Q102H))}; // SLT
    SLTU    : AluOutQ102H = {31'b0 , AluIn1Q102H < AluIn2Q102H};                  // SLTU
    // Shifter
    SLL     : AluOutQ102H = AluIn1Q102H << ShamtQ102H;                            // SLL
    SRL     : AluOutQ102H = AluIn1Q102H >> ShamtQ102H;                            // SRL
    SRA     : AluOutQ102H = $signed(AluIn1Q102H) >>> ShamtQ102H;                  // SRA
    // Bit wise operations
    XOR     : AluOutQ102H = AluIn1Q102H ^ AluIn2Q102H;                            // XOR
    OR      : AluOutQ102H = AluIn1Q102H | AluIn2Q102H;                            // OR
    AND     : AluOutQ102H = AluIn1Q102H & AluIn2Q102H;                            // AND
    default : AluOutQ102H = AluIn1Q102H + AluIn2Q102H;
  endcase
  if (CtrlLuiQ102H) AluOutQ102H = AluIn2Q102H;                                    // LUI
end
```

- Branch resolution:

```
always_comb begin : branch_comp
  // Check branch condition
  unique casez ({CtrlBranchOpQ102H})
    BEQ     : BranchCondMetQ102H =  (RegRdData1Q102H == RegRdData2Q102H);                 // BEQ
    BNE     : BranchCondMetQ102H = ~(RegRdData1Q102H == RegRdData2Q102H);                 // BNE
    BLT     : BranchCondMetQ102H =  ($signed(RegRdData1Q102H) < $signed(RegRdData2Q102H)); // BLT
    BGE     : BranchCondMetQ102H = ~($signed(RegRdData1Q102H) < $signed(RegRdData2Q102H)); // BGE
    BLTU    : BranchCondMetQ102H =  (RegRdData1Q102H < RegRdData2Q102H);                  // BLTU
    BGEU    : BranchCondMetQ102H = ~(RegRdData1Q102H < RegRdData2Q102H);                  // BGEU
    default : BranchCondMetQ102H = 1'b0;
  endcase
end

assign SelNextPcAluOutQ102H = (SelNextPcAluOutBQ102H && BranchCondMetQ102H)
                            || (SelNextPcAluOutJQ102H);
assign NextPcQ102H = SelNextPcAluOutQ102H ? AluOutQ102H : PcPlus4Q100H;
assign flushQ102H = SelNextPcAluOutQ102H;
```

Here SelNextPcAluOutQ102H checks if we are in branch instruction and the branch condition is met or are in jump instruction. If yes, NextPcQ102H get the calculated target address from AluOutQ102H and flushQ102H is on in that case.

3.3.4.4 *Memory (Q103H)*

- Access to RAM for reading and writing:

```
always_comb begin
DMemWrDataQ103H = (DMemAddressQ103H[1:0] == 2'b01 ) ? { RegRdData2Q103H[23:0],8'b0  } :
                  (DMemAddressQ103H[1:0] == 2'b10 ) ? { RegRdData2Q103H[15:0],16'b0 } :
                  (DMemAddressQ103H[1:0] == 2'b11 ) ? { RegRdData2Q103H[7:0] ,24'b0 } :
                                                        RegRdData2Q103H;
DMemByteEnQ103H = (DMemAddressQ103H[1:0] == 2'b01 ) ? { CtrlDMemByteEnQ103H[2:0],1'b0 } :
                  (DMemAddressQ103H[1:0] == 2'b10 ) ? { CtrlDMemByteEnQ103H[1:0],2'b0 } :
                  (DMemAddressQ103H[1:0] == 2'b11 ) ? { CtrlDMemByteEnQ103H[0]  ,3'b0 } :
                                                        CtrlDMemByteEnQ103H;
end
assign DMemAddressQ103H = AluOutQ103H;
assign DMemWrEnQ103H    = CtrlDMemWrEnQ103H;
assign DMemRdEnQ103H    = SelDMemWbQ103H;
```

Here we examine the lower 2 bits of the addresses we access and accordingly perform a non-cyclical shift to the information and the byte enable signals to enforce word alignment accesses to the memory. The shift operation was needed for supporting non 4 bytes align memory access due to HW constraint of the standard RAM which we used.

3.3.4.5 *Write Back (Q104H)*

- Selecting the information to be written back to the Register File - The ALU source or the memory source:

```
assign ByteOffsetQ104H = AluOutQ104H[1:0];

always_comb begin
ByteenaRestoreQ104H   = (ByteOffsetQ104H == 2'b01 ) ? { 1'b0,ByteEnQ104H[3:1] } : // we have done 1
shift - so 1 shift right
                        (ByteOffsetQ104H == 2'b10 ) ? { 2'b0,ByteEnQ104H[3:2] } : // we have done 2
shift - so 2 shift right
                        (ByteOffsetQ104H == 2'b11 ) ? { 3'b0,ByteEnQ104H[3]   } : // we have done 3
shift - so 3 shift right
                                                              ByteEnQ104H;         // we don't shifted
end
assign RdDataAfterShiftQ104H = (ByteOffsetQ104H == 2'b00) ?        DMemRdRspQ104H         :
                               (ByteOffsetQ104H == 2'b01) ? { 8'b0,DMemRdRspQ104H[31:8]}  :
                               (ByteOffsetQ104H == 2'b10) ? {16'b0,DMemRdRspQ104H[31:16]} :
                               (ByteOffsetQ104H == 2'b11) ? {24'b0,DMemRdRspQ104H[31:24]} :
                                                            DMemRdRspQ104H         ;
// Sign extend taking care of
assign PostSxDMemRdDataQ104H[7:0]   =  ByteenaRestoreQ104H[0] ? RdDataAfterShiftQ104H[7:0]     : 8'b0;
assign PostSxDMemRdDataQ104H[15:8]  =  ByteenaRestoreQ104H[1] ? RdDataAfterShiftQ104H[15:8]    :
                                       CtrlSignExtQ104H       ? {8{PostSxDMemRdDataQ104H[7]}}  : 8'b0;
assign PostSxDMemRdDataQ104H[23:16] =  ByteenaRestoreQ104H[2] ? RdDataAfterShiftQ104H[23:16]   :
                                       CtrlSignExtQ104H       ? {8{PostSxDMemRdDataQ104H[15]}} : 8'b0;
assign PostSxDMemRdDataQ104H[31:24] =  ByteenaRestoreQ104H[3] ? RdDataAfterShiftQ104H[31:24]   :
                                       CtrlSignExtQ104H       ? {8{PostSxDMemRdDataQ104H[23]}} : 8'b0;

// ---- Select what write to the register file ----
assign WrBackDataQ104H = SelDMemWbQ104H  ? PostSxDMemRdDataQ104H : AluOutQ104H;
assign RegWrDataQ104H  = SelRegWrPcQ104H ? PcPlus4Q104H          : WrBackDataQ104H;
```

Here we do some interesting things. First, we undo the operation of the shift that we performed in the previous step so that we do not allow crossword reads. We handle the sign bit according to the type of instruction - signed or unsigned and choose what to write back to the Register File - the ALU source or the memory source.

The Figure below shows the complete architecture of the RVC_ASAP core:

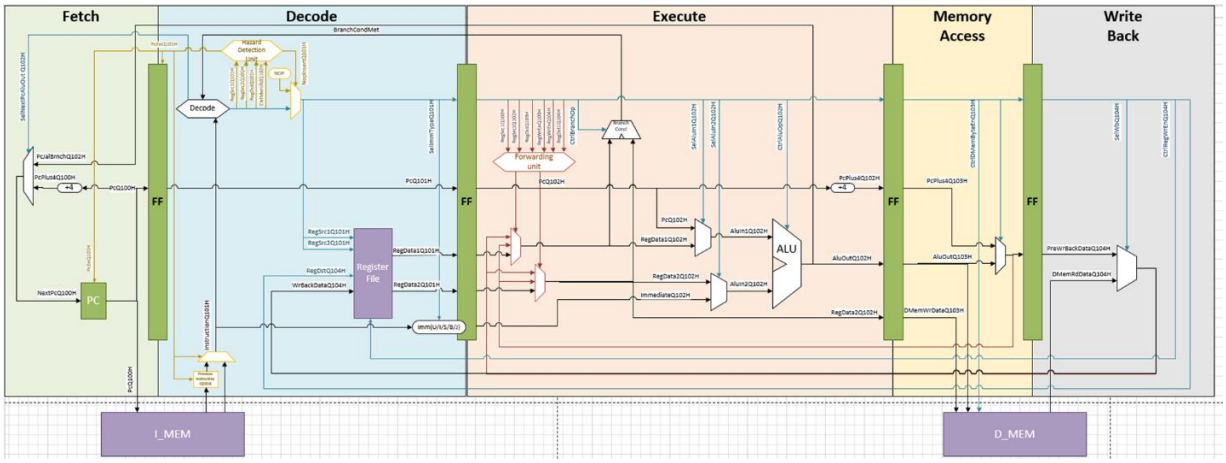

Figure 7 - RVC_ASAP detailed block diagram

3.3.5 Memory regions

The division of the memory space from the point of view of the process is more detailed. It doesn't differ in its general delimitation limits although the memory of the process requires a further division of the memory spaces into additional spaces. The memory space of the processor in our case is equivalent to the memory space of a single process since there is always one and only one process in the system. The division of the memory space of the process is carried out by the linker script. We touch on this topic in the GCC section.

At this point, we will expand on the memory components as they are defined in hardware.

3.3.5.1 mem_wrap

The Memory Wrapper wraps together the Instruction Memory, Data Memory, CR Memory, and VGA Controller. The Figure below illustrates this interface:

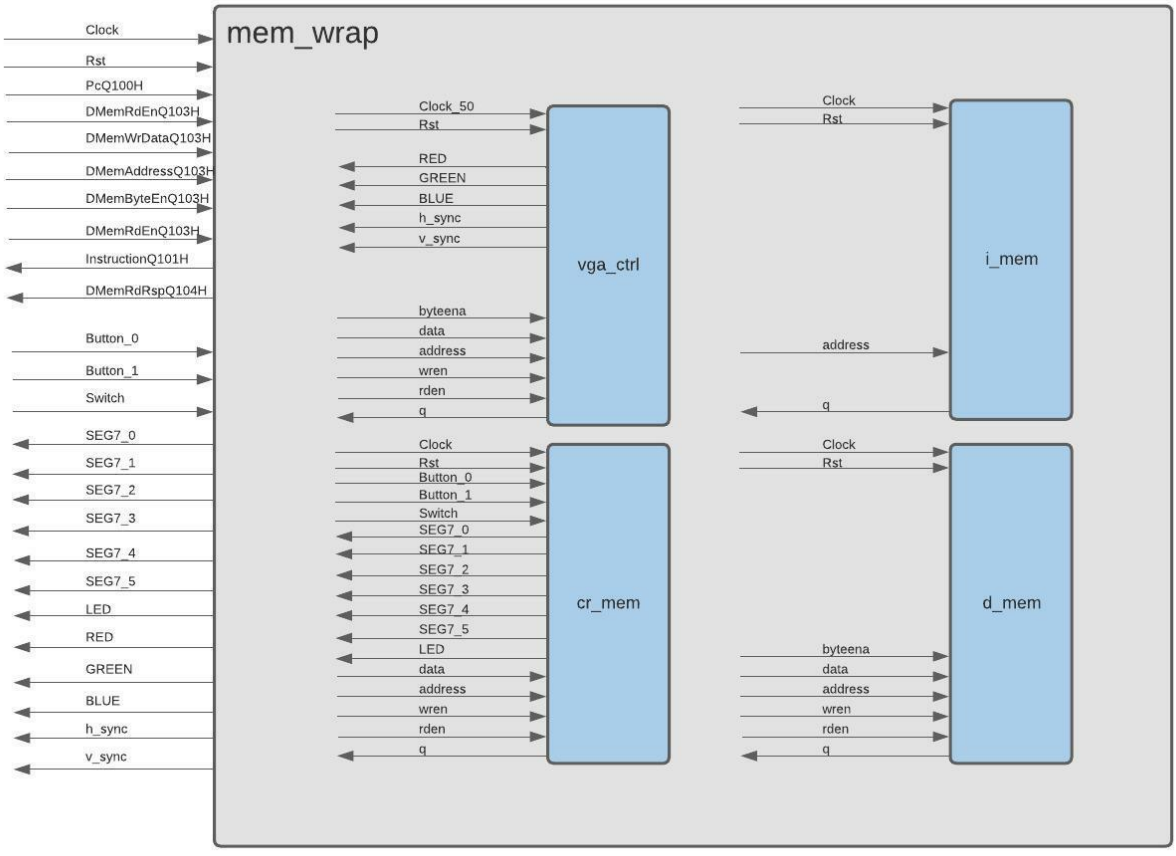

Figure 8 - mem_wrap interface

Signal	Size [bits]	Description
Clock	1	Clock signal.
Rst	1	Reset signal.
Button_0	1	Key 0.
Button_0	1	Key 1.
Switch	10	The 10 switch buttons.
SEG7_0	8	The 8 bulbs of segment 0.
SEG7_1	8	The 8 bulbs of segment 1.
SEG7_2	8	The 8 bulbs of segment 2.
SEG7_3	8	The 8 bulbs of segment 3.
SEG7_4	8	The 8 bulbs of segment 4.
SEG7_5	8	The 8 bulbs of segment 5.
LED	10	The 10 led bulbs.
RED	4	The red signal of the VGA.
GREEN	4	The green signal of the VGA.
BLUE	4	The blue signal of the VGA.
h_sync	1	Horizontal synchronization signal of the VGA.
V_sync	1	Vertical synchronization signal of the VGA.
PreInstructionQ101H	32	Pre instruction to decode.
DMemRdRspQ104H	32	Data to read from the memory.
PcQ100H	32	Program Counter.
DMemWrDataQ103H	32	Data to write to the memory.
DMemAddressQ103H	32	Address.
DMemByteEnQ103H	4	Byte enable – defines which bytes to choose from the four bytes.
DMemWrEnQ103H	1	Write enable.
DMemRdEnQ103H	1	Read enable.

Table 6 - mem_wrap interface signals

The technique in which we assign the address that comes out of the core towards the mem_wrap to the memory space relevant to the address is shown in the following code segment:

```
always_comb begin
    MatchVGAMemRegionQ103H = ((DMemAddressQ103H[VGA_MSB_REGION:LSB_REGION] >= VGA_MEM_REGION_FLOOR)
                              && (DMemAddressQ103H[VGA_MSB_REGION:LSB_REGION] <= VGA_MEM_REGION_ROOF));
    MatchDMemRegionQ103H   = MatchVGAMemRegionQ103H ? 1'b0 :
                             ((DMemAddressQ103H[MSB_REGION:LSB_REGION]  >= D_MEM_REGION_FLOOR)
                             && (DMemAddressQ103H[MSB_REGION:LSB_REGION] <= D_MEM_REGION_ROOF));
    MatchCRMemRegionQ103H  = MatchVGAMemRegionQ103H ? 1'b0 :
                             ((DMemAddressQ103H[MSB_REGION:LSB_REGION] >= CR_MEM_REGION_FLOOR)
                             && (DMemAddressQ103H[MSB_REGION:LSB_REGION] <= CR_MEM_REGION_ROOF));
end
`RVC_MSFF(MatchDMemRegionQ104H   , MatchDMemRegionQ103H    , Clock) // Q103H to Q104H Flip Flops
`RVC_MSFF(MatchCRMemRegionQ104H  , MatchCRMemRegionQ103H   , Clock) // Q103H to Q104H Flip Flops
`RVC_MSFF(MatchVGAMemRegionQ104H , MatchVGAMemRegionQ103H  , Clock) // Q103H to Q104H Flip Flops
// Mux between CR ,data and vga memory
assign DMemRdRspQ104H= MatchCRMemRegionQ104H  ? PreCRMemRdDataQ104H  :
                       MatchDMemRegionQ104H   ? PreDMemRdDataQ104H   :
                       MatchVGAMemRegionQ104H ? PreVGAMemRdDataQ104H :
                                                32'b0                ;
```

We check the memory area to which the address belongs explicitly and turn on the signals MatchDMemRegionQ103H, MatchCRMemRegionQ103H, MatchVGAMemRegionQ103H accordingly. In the next step, we send to each memory component in the *rden* and *wren* ports the result of the AND operation between each signal and the match region corresponding to it and in this way only the component that needs to work works. One clock cycle ahead, we check against which memory area we worked and return the corresponding port DMemRdRspQ104H.

3.3.5.2 I_MEM

I_MEM is the memory region that contains the instructions of the program. It is S-RAM based and its size is 16kb. The program is loaded to the I_MEM with backdoor load through the Test Bench if we are in simulation or as setting the initial memory content if we are in FPGA. The memory size is control by the parameter I_MEM_MSB and its basic size is 16kb. The access to I_MEM is a single-port access. The processor accesses I_MEM in the FETCH stage. In the simulation we used behavioral memory, and on the FPGA, we used the on-die FPGA memory. In addition, on the FPGA we used the MIF format to pre-load the I_MEM as a ROM.

3.3.5.3 D_MEM

D_MEM is the memory region that contains the data of the program. It is S-RAM based and its size is 16kb. The program data is loaded to the D_MEM with backdoor load through the Test Bench if we are in simulation or as setting the initial memory content if we are in FPGA. The memory size is control by the parameter D_MEM_MSB and its basic size is 16kb. The access to D_MEM is a single-port access. The processor accesses D_MEM in the Memory stage. In the simulation we used behavioral memory, and on the FPGA, we used the on-die FPGA memory. In addition, on the FPGA we used the MIF format to pre-load the D_MEM which then can be overwritten by the SW (RAM).

3.3.5.4 CR_MEM

CR_MEM is the control registers memory region. It is a flip-flop based that contains essential registers for the communication with the FGPA peripherals. CR data can be accessed through the software by requesting the specific offset of the specific CR in the CR table. Some CRs are read only that sample data from the hardware. Others are read and write CRs. The memory size is control by the parameter CR_MEM_MSB and its basic size is 4kb. The access to CR_MEM is a single-port access. The processor accesses CR_MEM in the Memory stage. The table below shows the control registers present in the design:

CR Name	Address	Description
CR_SEG7_0	0x00007000	RW 7 bit.
CR_SEG7_1	0x00007004	RW 7 bit.
CR_SEG7_2	0x00007008	RW 7 bit.
CR_SEG7_3	0x0000700C	RW 7 bit.
CR_SEG7_4	0x00007010	RW 7 bit.
CR_SEG7_5	0x00007014	RW 7 bit.
CR_LED	0x00007018	RW 10 bit.
CR_Button_0	0x0000701C	RO 1 bit.
CR_Button_1	0x00007020	RO 1 bit.
CR_Switch	0x00007024	RO 10 bit.
CR_CURSOR_H	0x00007028	RW 32 bit.
CR_CURSOR_V	0x0000702C	RW 32 bit.

Table 7 - Control Registers

To get the full understanding behind each control register, look at Figure 9. The names of the control registers correspond to the names indicated in the figure.

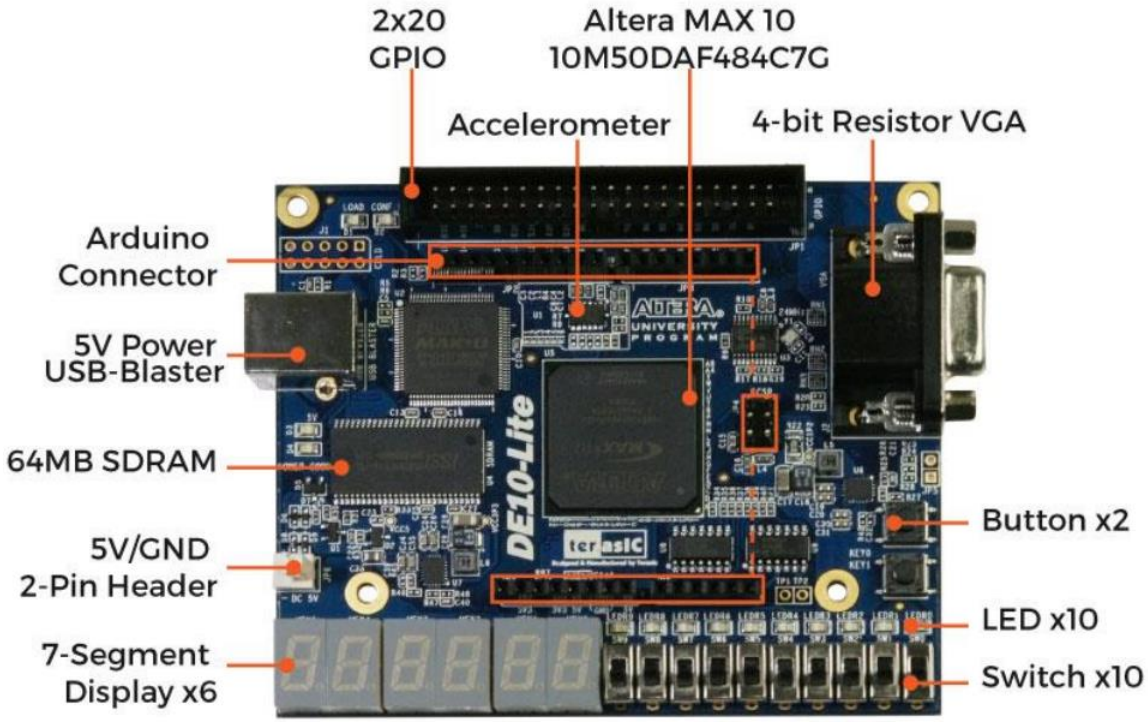

Figure 9 - DE10-Lite intel FPGA [image from DE10-Lite User Manual]

The way we implemented CR_MEM is the following way. First, we defined 2 structs:

```
typedef struct packed { // RO
    logic       Button_0;
    logic       Button_1;
    logic [9:0] Switch;
} t_cr_ro ;

typedef struct packed { // RW
    logic [7:0]  SEG7_0;
    logic [7:0]  SEG7_1;
    logic [7:0]  SEG7_2;
    logic [7:0]  SEG7_3;
    logic [7:0]  SEG7_4;
    logic [7:0]  SEG7_5;
    logic [9:0]  LED;
    logic [31:0] CR_CURSOR_H;
    logic [31:0] CR_CURSOR_V;
} t_cr_rw ;
```

The idea behind the definition of the structures is to integrate the control registers in a convenient and simple way while dividing into control registers which can only be read and those which can be both read and written.

In the next step we defined the memory as follows:

```
t_cr_ro cr_ro;
t_cr_rw cr_rw;
t_cr_ro cr_ro_next;
t_cr_rw cr_rw_next;
// Data-Path signals
logic [31:0] pre_q;
always_comb begin
    cr_ro_next = cr_ro;
    cr_rw_next = cr_rw;
    if(wren) begin
        unique casez (address) // address holds the offset
            // ---- RW memory ----
            CR_SEG7_0   : cr_rw_next.SEG7_0         = data[7:0];
            CR_SEG7_1   : cr_rw_next.SEG7_1         = data[7:0];
            CR_SEG7_2   : cr_rw_next.SEG7_2         = data[7:0];
            CR_SEG7_3   : cr_rw_next.SEG7_3         = data[7:0];
            CR_SEG7_4   : cr_rw_next.SEG7_4         = data[7:0];
            CR_SEG7_5   : cr_rw_next.SEG7_5         = data[7:0];
            CR_LED      : cr_rw_next.LED            = data[9:0];
            CR_CURSOR_H : cr_rw_next.CR_CURSOR_H    = data[31:0];
            CR_CURSOR_V : cr_rw_next.CR_CURSOR_V    = data[31:0];
            // ---- Other ----
            default   : /* Do nothing */;
        endcase
    end
```

This is a partial view of the memory code because it is long, but the view is enough to understand the principles behind it. First, we instantiate each of the structures we defined earlier. In the case shown in the code section, we perform a write. If the address matches one of the addresses of any CR, we update the value of the CR represented as a variable in the appropriate struct accordingly.

3.3.5.5 VGA_MEM

VGA_MEM is the memory region that responsible for communicating with the screen. It is also S-RAM based and its size is 32.5kb. Any writing to a certain area in this memory region will be reflected to the screen and in addition to that the processor can read the status of a certain pixel assuming that the software is interested in it. There is an option for this memory area to be initialized to zero by the software that run on the processor if it so wishes. The memory size is control by the parameter VGA_MEM_MSB and its basic size is 32.5kb. The access to VGA_MEM is a dual-port access because both the processor and the VGA controller can access this memory area together. The processor accesses VGA_MEM in the Memory stage. The driver that pulls the information from this memory region to the screen is the VGA Controller. In the simulation we used behavioral memory, and on the FPGA, we used the on-die FPGA memory (as same as I_MEM and D_MEM).

All the memory regions that have been described so far are wrapped by mem_wrap so that at the level of this component the address is checked, and the reference is made to the relevant memory space. In relation to VGA_MEM, this memory component is wrapped by the VGA Controller and the VGA Controller is a component of mem_wrap. That is, every request to the memory area of the VGA goes through the processor, passes through the mem_wrap to the VGA Controller and then reaches the memory area of the VGA.

Now, we will touch in detail on the VGA Controller.

3.3.6 VGA Controller

The resolution of our target screen is 640×480. As can be seen from Figure 10, there are 640 vertical bits lines and 480 horizontal bits lines. Overall, the screen contains 80×480 = 38,400 bytes, and if we take into account that the size of a word is 4 bytes, the screen contains 9600 words. Each pixel on the screen is represented by a single bit, as a result, the total number of pixels is 640×480 = 307,200 pixels. The VGA support 12 bit RGB, but we decided that having such a large amount of memory was a waste. So, we implemented a monochromatic screen, which each pixel is either on or off.

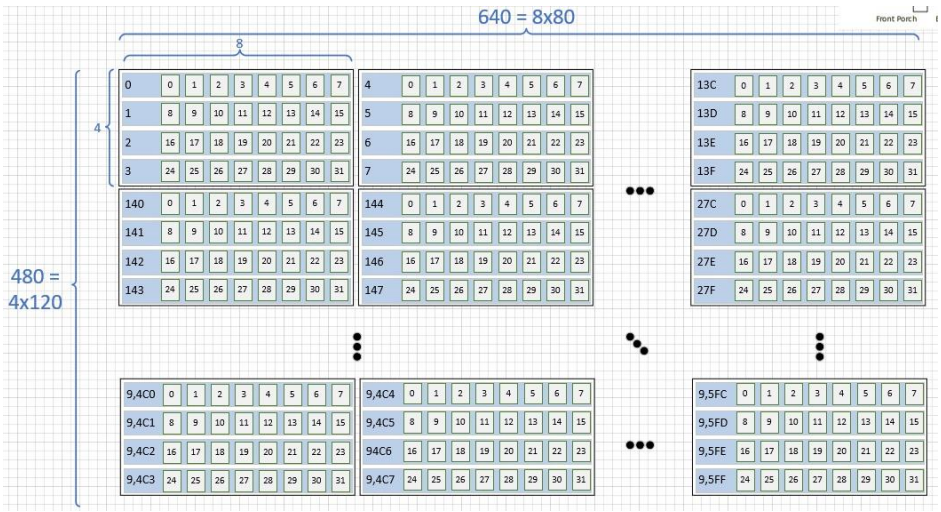

Figure 10 - The screen

VGA at 640×480 resolution at 60Hz (frames per second) requires a pixel clock of 25.175Mhz, this is the industry standard. The tables below show the necessary timing for the VGA control signals at 640×480 resolution at 60Hz. You can view additional timings for different cases in the following link.

General timing

Parameter	Timing
Screen refresh rate	60 HZ
Vertical refresh	31.46875 kHz
Pixel freq.	25.175 MHz

Table 8 - VGA general timing

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

Scanline part	Pixels	Time [µs]
Visible area	640	25.422045680238
Front porch	16	0.63555114200596
Sync pulse	96	3.8133068520357
Back porch	48	1.9066534260179
Whole line	800	31.777557100298

Table 9 - VGA horizontal timing

Vertical timing (frame)

Polarity of vertical sync pulse is negative.

Frame part	Lines	Time [ms]
Visible area	480	15.253227408143
Front porch	10	0.31777557100298
Sync pulse	2	0.063555114200596
Back porch	33	1.0486593843098
Whole line	525	16.683217477656

Table 10 - VGA vertical timing

To understand how the transmission to the screen is carried out, the first two signals that deserve mention and explanation are the horizontal and vertical synchronization signals.

Horizontal Synchronization

Figure 11 shows the wave diagram of the horizontal synchronization signal. We will look at the hsync (horizontal synchronization) signal and how it relates to how the electron beam of the CRT monitor traverses across the CRT monitor. In Figure 11 we can see a CRT monitor with a black border, and the reason is because older CRT monitors did have that black border and it is part of the scanning pattern that we do. That is, we do pass the electron beam across the black border too, however we don't show any pixels over there.

The hsync signal is a periodic digital signal that goes between zero and one logic. Based on the value of that signal the CRT monitor generates a sawtooth signal which moves the electron beam across the screen. The period of hsync signal is between 0 all the way to 799, which is 800. Suppose we start at the point marked 0 in the wave diagram of the hsync signal in Figure 11, in principle we can start at any point we want because the signal is periodic. Point 0 is the start of the screen, this point located in the visible portion of the screen, not the border. We set the hsync signal to 1 all the way to 639 which means the electron beam will travers across the CRT monitor from left to right. We can observe that the sawtooth signal in this section is going all the way up, but it didn't finish though. What is happens next is that the electron beam goes through the right border (front porch) for exactly 16 clock cycles (not the system clock cycle). Then, the horizontal line must go backwards for 96 clock cycles, this is what we call the retrace and the sawtooth signal in this section is going all the way down to zero. In the final stage, relative to the point where we started (0) and because the signal is periodic the electron beam goes through the left border (back porch) for exactly 48 clock cycles. Realization of the hsync signal is basically by a simple counter which goes from 0 to 799 (modulo 800).

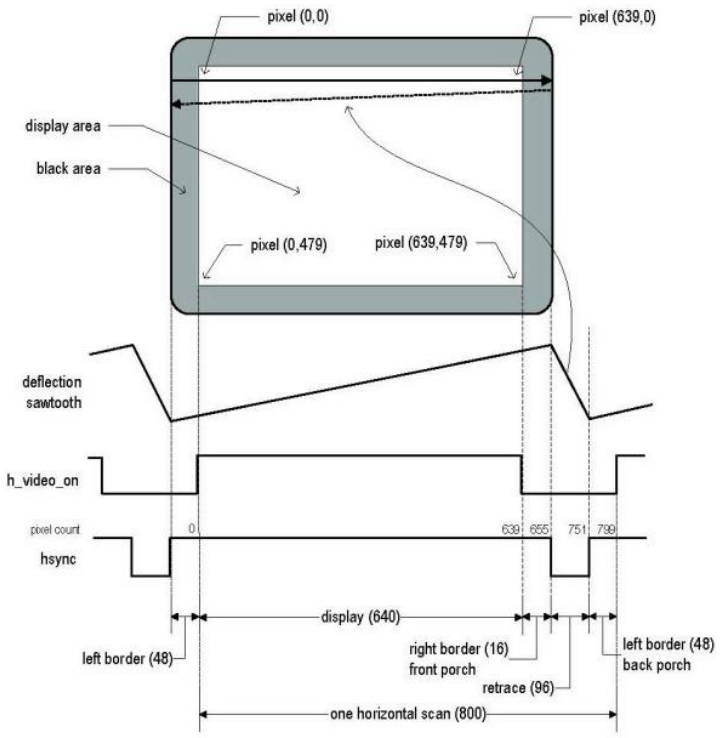

Figure 11 - Horizontal Synchronization

Vertical Synchronization

Figure 12 shows the wave diagram of the vertical synchronization signal. Now, we will look at the vsync (vertical synchronization) signal and how it relates to how the electron beam of the CRT monitor traverses across the CRT monitor. The horizontal synchronization signal allowed us to move the electron beam or the scanning of the pixels from left to right, so we need another signal which goes from top to bottom, this is where the vertical synchronization signal comes into play. We want to go from top to bottom in 1/60 seconds because we have a refresh rate of 60Hz.

vsync signal is very similar to hsync signal so the vsync signal is also a periodic digital signal that goes between zero and one logic. Suppose we start at the point marked 0 in the wave diagram of the vsync signal in Figure 12, here also we can start at any point we want because the signal is periodic. Point 0 is the start of the screen, this point located in the visible portion of the screen, not the border. We set the vsync signal to 1 all the way to 479 which means the electron beam will travers across the CRT monitor from top to bottom. What is happens next is that the electron beam goes through the bottom border (front porch) for exactly 10 clock cycles (not the system clock cycle). Then, the vertical line must go backwards for 2 clock cycles, this is the retrace. In the final stage, relative to the point where we started (0) and because the signal is periodic the electron beam goes through the top border (back porch) for exactly 33 clock cycles. Realization of the vsync signal is also basically by a simple counter which goes from 0 to 524 (modulo 525).

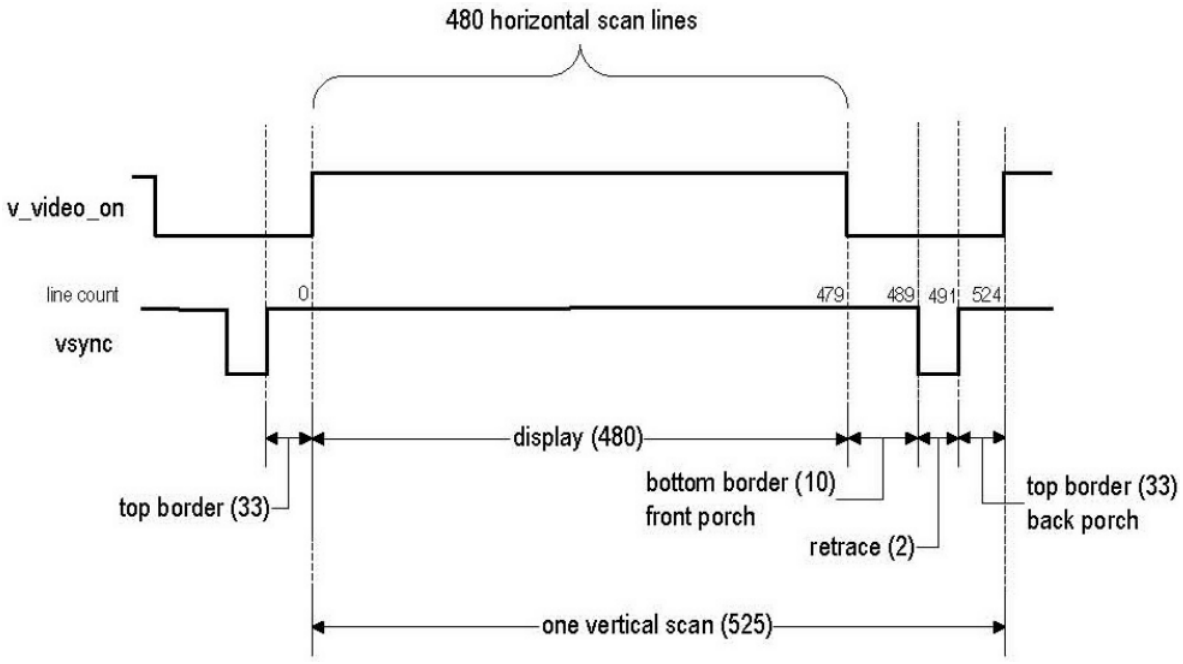

Figure 12 - Vertical Synchronization

Now, let's look at the complete picture. That is, let's consider the horizontal and vertical synchronization signals together. Figure 13 shows the complete scanning pattern of the screen.

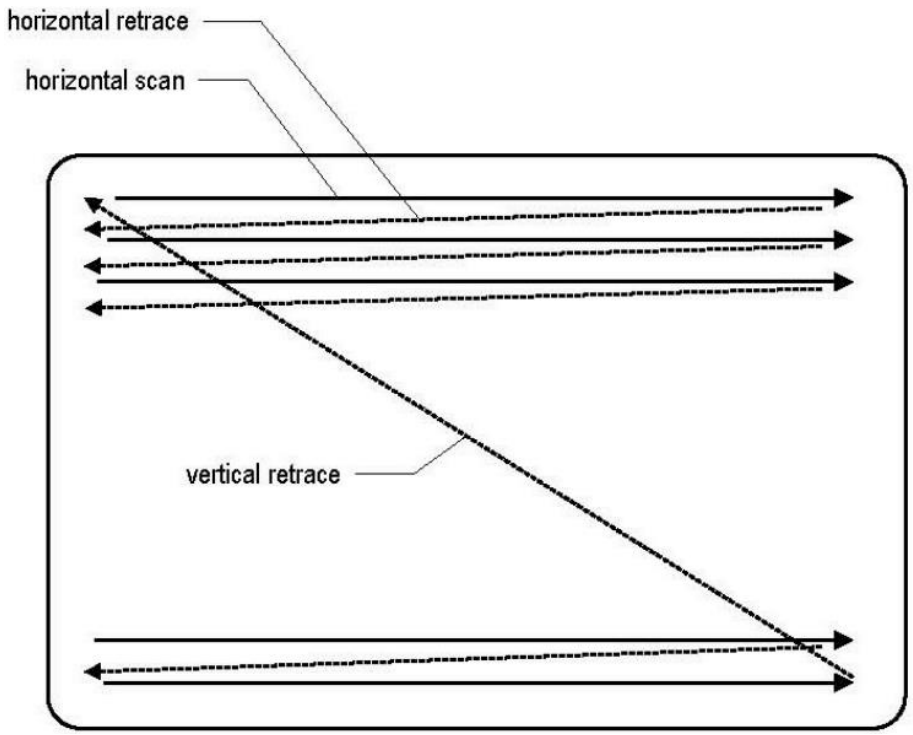

Figure 13 - Screen scanning pattern

As mentioned in the HAS section, The VGA Controller contains the following instances:

- VGA Synchronization Machine (sync_gen).
- 25MHz Clock Generator (pll_2).
- Video Graphic Array Memory (VGA_MEM).

3.3.6.1 sync_gen

This module is the controller of the hsync and vsync signals:

```
// VGA @ 640x480 resolution @ 60Hz requires a pixel clock of 25.175Mhz.
// Maxed x = 800 , y = 525
assign CounterXmaxed = (CounterX == 800) || Reset; // 16 + 48 + 96 + 640
assign CounterYmaxed = (CounterY == 525) || Reset; // 10 + 2 + 33 + 480

// x and y counters
`RVC_RST_MSFF   (CounterX, (CounterX+1'b1), CLK_25, CounterXmaxed)
`RVC_EN_RST_MSFF(CounterY, (CounterY+1'b1), CLK_25, CounterXmaxed, (CounterXmaxed && CounterYmaxed) )
assign next_h_sync = (CounterX >= (640 + 16) && (CounterX < (640 + 16 + 96)));  // active for 96
clocks
assign next_v_sync = (CounterY >= (480 + 10) && (CounterY < (480 + 10 + 2)));   // active for 2 clocks

`RVC_MSFF(h_sync, next_h_sync, CLK_25)
`RVC_MSFF(v_sync, next_v_sync, CLK_25)

// Indication that we must not send Data in VGA RGB
assign NextinDisplayArea = ((CounterX < 640) && (CounterY < 480));
`RVC_MSFF(inDisplayArea, NextinDisplayArea, CLK_25)
assign vga_h_sync = ~h_sync;
assign vga_v_sync = ~v_sync;
```

The signal ConterXmaxed resets or when Reset signal is on or when CounterX signal reaches the value of 800. The value 800 is derived from 16 clock cycles in the front porch area, 96 clock cycles retrace, 48 clock cycles left border and 640 clock cycles display. In a similar way, the signal CounterYmaxed resets or when Reset signal is on or when CounterY signal reaches the value of 525. The value 525 is derived from 10 clock cycles in the front porch area, 2 clock cycles retrace, 33 clock cycles back porch and 480 clock cycles display.

3.3.6.2 Pll_2

This module is a frequency divider:

```
`RVC_RST_MSFF(CLK_25, !CLK_25, CLK_50, Reset)
```

The exact pixel frequency of the VGA for the resolution we used is 25.175MHz. The system clock frequency is 50MHz. Instead of using PLL we simply divide the system clock by 2 to get 25Mhz clock and its works fine as a clock of the VGA (although it's not exactly accurate).

3.3.6.3 VGA_MEM

Explained in VGA_MEM section.

4 VERIFICATION PLAN (HW)

4.1 INTRODUCTION

Validation is one of the most important procedures at product manufacturing and quality assurance. Validation definition is the assurance that a product, system, or service is working as expected. For example, a very familiar validation is the post-silicon validation, which is the performance, functionality, power, voltage, and more verifying tests after the CPU is manufactured physically. In RVC_ASAP project validation was critical to ensure the correctness of RVC_ASAP core and memory functionality, before continuing to the next project step, the FPGA synthesis. On our first step, when we still do not have a physical RVC_ASAP unit, we checked RVC_ASAP functionality with ModelSim simulations on a Test Bench.

4.2 TEST BENCH

Test Bench is an environment used to verify the correctness or soundness of a design or model. In our project, the Test Bench instantiate the top view of the design. The Test Bench also provides clock and reset signals to the design, loads the instruction and the data memories by a backdoor load and monitors them.

The backdoor load for the Instruction and Data Memory occurs in the following code snippet:

```
initial begin: test_seq
    //=====================================
    //load the program to the TB
    //=====================================
    $readmemh({"../apps/sv/",hpath,"-inst_mem_rv32i.sv"}, IMem);
    $readmemh({"../apps/sv/",hpath,"-data_mem_rv32i.sv"}, DMem);
    // Backdoor load the Instruction memory
    force rvc_asap_5pl_tb.rvc_top_5pl.rvc_asap_5pl_mem_wrap.rvc_asap_5pl_i_mem.IMem = IMem; //XMR
    assign Ebrake = rvc_asap_5pl_tb.rvc_top_5pl.rvc_asap_5pl.InstructionQ101H; //XMR
    // Backdoor load the data memory
    force rvc_asap_5pl_tb.rvc_top_5pl.rvc_asap_5pl_mem_wrap.rvc_asap_5pl_d_mem.DMem = DMem; //XMR
    # 10
    release rvc_asap_5pl_tb.rvc_top_5pl.rvc_asap_5pl_mem_wrap.rvc_asap_5pl_d_mem.DMem;
    #1000000
    end_tb("====TEST TIME OUT====");
end: test_seq
```

Code Snippet 1 - Backdoor load

At the end of the TB code there is a task that initiated by EBRAKE instruction and ends the simulation by creating a snapshot of the data memory and extracting the snapshot to a log file. In addition, a snapshot of the VGA memory region is taken to simulate screen printing which also extracted to a log file.

4.3 CHECKER

In the final stage we check the results by conduct a comparison between actual output of the test to the memory and its golden image. Golden image is pre-generated memory image produced from Venus RISC-V simulator. This Visual Studio Code extension embeds the popular Venus RISC-V simulator. It provides a standalone learning environment as no other tools are needed. It runs RISC-V assembly code with the standard debugging capabilities of VS Code. Here is a link for more information about the Venus RISC-V simulator. The golden image was extracted from the Venus RISC-V simulator by run and debug the Assembly code in the simulator. Venus simulator provides a way to view the memory dump after running the program. In this way we could build the golden image (correct memory snapshot) after running the program.

The Test Bench builds a memory snapshot that corresponds to the format of the golden image as follows:

```
    // Data memory snapshot
    fd = $fopen({"../target/",hpath,"/mem_snapshot.log"},"w");
    if (fd) $display("File was open succesfully : %0d", fd);
    else $display("File was not open succesfully : %0d", fd);
    for (i = 0 ; i < SIZE_D_MEM; i = i+4) begin
        $fwrite(fd,"Offset %08x : %02x%02x%02x%02x\n",i+D_MEM_REGION_FLOOR,
rvc_asap_5pl_tb.rvc_top_5pl.rvc_asap_5pl_mem_wrap.rvc_asap_5pl_d_mem.DMem[i+D_MEM_REGION_FLOOR+3],
rvc_asap_5pl_tb.rvc_top_5pl.rvc_asap_5pl_mem_wrap.rvc_asap_5pl_d_mem.DMem[i+D_MEM_REGION_FLOOR+2],
rvc_asap_5pl_tb.rvc_top_5pl.rvc_asap_5pl_mem_wrap.rvc_asap_5pl_d_mem.DMem[i+D_MEM_REGION_FLOOR+1],
rvc_asap_5pl_tb.rvc_top_5pl.rvc_asap_5pl_mem_wrap.rvc_asap_5pl_d_mem.DMem[i+D_MEM_REGION_FLOOR]);
    end
    $fclose(fd);
```

Code Snippet 2 - Test Bench memory snapshot

As part of the automation process, the buildl.sh script compares the two files (golden image and real image) and alerts if there is a discrepancy between the two files and where the discrepancy exists.

In addition, the Test Bench builds a VGA Memory snapshot as follows:

```
    // VGA memory snapshot - simulate a screen
    fd1 = $fopen({"../target/",hpath,"/screen.log"},"w");
    if (fd1) $display("File was open succesfully : %0d", fd1);
    else $display("File was not open succesfully : %0d", fd1);
    for (i = 0 ; i < 38400; i = i+320) begin // Lines
        for (j = 0 ; j < 4; j = j+1) begin // Bytes
            for (k = 0 ; k < 320; k = k+4) begin // Words
                for (l = 0 ; l < 8; l = l+1) begin // Bits
                    if
(rvc_asap_5pl_tb.rvc_top_5pl.rvc_asap_5pl_mem_wrap.rvc_asap_5pl_vga_ctrl.rvc_asap_5pl_vga_mem.VGAMem[k+j+i][l])
$fwrite(fd1,"X");
                    else $fwrite(fd1," ");
                end
            end
            $fwrite(fd1,"\n");
        end
    end
    $fclose(fd1);
```

Code Snippet 3 - Test Bench VGA memory snapshot

This was helpful to enable the VGA mem in simulation.

5 FPGA

In this section we will present several statistics from the utilization of the FPGA component after uploading our design onto it.

Flow summary:

Parameter	Value
Flow Status	Successful - Thu Sep 15 11:15:24 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	CPU_GARAGE
Top-level Entity Name	Top
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	3,540 / 49,760 (7%)
Total combinational functions	3,283 / 49,760 (7%)
Dedicated logic registers	1,706 / 49,760 (3%)
Total registers	1706
Total pins	85 / 360 (24%)
Total virtual pins	0
Total memory bits	569,437 / 1,677,312 (34 %)
Embedded Multiplier 9-bit elements	0 / 288 (0%)
Total PLLs	1 / 4 (25%)
UFM blocks	0 / 1 (0%)
ADC blocks	0 / 2 (0%)

Table 11 - FPGA statistics

As can be seen from the statistics presented, we use 34% of the total memory that can be used on the FPGA. As a result, there is the option to significantly increase the instruction memory and the data memory and build larger and more complex programs.

6 SW & API

6.1 RISC-V GNU Compiler Toolchain

The Toolchain is a bundle of software tools cloned from riscv-gnu-toolchain Git repositories. It was used to create and verify assembly instructions against the open-source ISA specification for an RV32IM core. A detailed installation guide can be found in this link on John Winans repository.

6.2 GCC

In simplicity, the Toolchain contains several of special compilers based on the famous GCC compiler family and we use it to compile a *.c* file written in C language to a RISC-V Assembly language file (*.s*), and from that file, to a binary *.sv* file that will simulate the Instruction Memory such that RVC_ASAP can read. The toolchain also generates these files as text files that the user can read. The Assembly code is longer than the C code. Each Assembly command is coded in a 32-bit vector (or 8 hexadecimal digit vector in the text file). All vectors combined to form the simulation Instruction Memory, which can be seen in Code Snippet 15. The RVC_ASAP decodes each vector. The SV text file will used as an Instruction Memory to the RVC_ASAP in the simulations via Back Door technique which discussed in here.

After writing a C program that we want to run on the RVC_ASAP Test Bench, and after considering the expected results for the program, we use the RISC-V Toolchain commands on our local Windows machine in Git Bash terminal. Before we proceed further, there are several important elements that we need to go over.

- **RVC_ASAP Initialization Assembly code file** – a file named *crt0.s* which contains a code written in RISC-V Assembly that supposed to run on the RVC_ASAP Test Bench before any C program. It has those main stages:
    - Pipeline cleanup – 5 NOP (No Operation) instructions entered to RVC_ASAP to clean the pipe from previous or undefined data.
    - Initialize registers – Set all 32 core registers to 0 using *mv* instruction and register *x0* which always contains zero value.
    - Stack initialization and CR initialization – Load to the Stack Pointer (SP) the address of the Stack by the following command:
      la sp, _stack
      _stack parameter is defined by the linker script. Moreover, initialize the CR_CURSOR_H and CR_CURSOR_V which points to the current place of the cursor by the following commands:
      li x31, 0x7028 # CR_CURSOS_H address
      mv x1, x0 # Reset x1 register
      sw x1, 0x0(x31) # Reset CR_CURSOR_H
      sw x1, 0x4(x31) # Reset CR_CURSOR_V
    - Jump to main – Jump directly to the C program main function.
    - Terminate run – after returning from C program's main, we use EBREAK instruction to terminate the Test Bench.

Linker script file *– An LD file, which is a script written in the GNU Linker Command Language. The Toolchain use the linker to link the crt0.s code and the program Assembly code to a single Assembly program. It also defines the sizes and origin addresses of the Instruction and Data Memories and the Stack Memory. The central part of the linker script is shown in*

- Code Snippet 4. In this section we determine the memory regions as they are observed from the perspective of the process: *instrram*, *dataram*, and *stack*. Those regions are further divided into internal sub-regions.

```
MEMORY
{
    instrram    : ORIGIN = 0x00000000, LENGTH = 0x4000
    dataram     : ORIGIN = 0x00004000, LENGTH = 0x3000
    stack       : ORIGIN = 0x00006E00, LENGTH = 0x0200
}
```

Code Snippet 4 - Memory definition inside the linker script

The further division occurs as follows:

- *instrram* is divided to:
  - *vectors* – Contains ISR vector table.
  - *text* – Contains executable code and is generally read-only and fixed size.
- *dataram* is divided to:
  - *rodata* – Contains initialized static variables, i.e., global variables and local static variables which have a defined value and cannot be modified.
  - *data* – Contains initialized static variables, i.e., global variables and local static variables which have a defined value and can be modified.
  - *bss* - Contains uninitialized static data, both variables and constants, i.e., global variables and local static variables that are initialized to zero or do not have explicit initialization in source code.

Because the access to the memory in our architecture is word-aligned (4 bytes) we are enforcing the same alignment in placing the data into the memory, we do that by the ALIGN (4) instruction as follow:

```
    .rodata : {
        . = ALIGN(4);
        *(.rodata);
        *(.rodata.*)
    } > dataram
```

Code Snippet 5 - Enforce data alignment by the linker

Now, Figure 3 from the HAS section looks from the point of view of the process as follows:

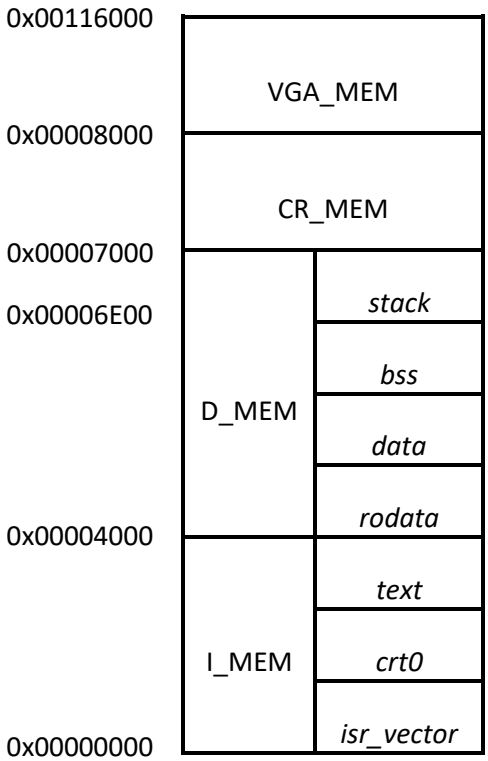

Figure 14 - Address space from process's point of view

It is worth noting that in our case we do not use the *heap* section, but of course it is possible to use it and even implement and use the malloc function.
To read more about linker please visit here.

- **Assembler** – An Assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions Assembler language and others use the term Assembly language.

6.3 HW API

The hardware resource APIs allow you to work with hardware resources. A hardware resource is an addressable piece of hardware on the system. A hardware resource is known to the system by its resource name. A resource entry is the reference to the hardware resource in the hardware resource information, which can be thought of as a list of the hardware resources on the system [from IBM.com].

In our project the basis of the communication through the software with the hardware, thereby substituting the HW API, is the defines which are defined in the file *rvc_defines.h*. This file contains the following settings:

```
#define D_MEM_BASE    0x00004000
#define CR_MEM_BASE   0x00007000
#define VGA_MEM_BASE 0x00008000
#define FP_RESULTS    0x00004f00

#define WRITE_REG(REG,VAL) (*REG) = VAL
#define READ_REG(VAL,REG)  VAL    = (*REG)
#define MEM_SCRATCH_PAD    ((volatile int *) (D_MEM_BASE))
#define MEM_SCRATCH_PAD_FP  ((volatile float *) (FP_RESULTS))

/* Control registers addresses */
#define CR_SEG7_0   (volatile int *) (CR_MEM_BASE + 0x0)
#define CR_SEG7_1   (volatile int *) (CR_MEM_BASE + 0x4)
#define CR_SEG7_2   (volatile int *) (CR_MEM_BASE + 0x8)
#define CR_SEG7_3   (volatile int *) (CR_MEM_BASE + 0xc)
#define CR_SEG7_4   (volatile int *) (CR_MEM_BASE + 0x10)
#define CR_SEG7_5   (volatile int *) (CR_MEM_BASE + 0x14)
#define CR_LED      (volatile int *) (CR_MEM_BASE + 0x18)
#define CR_Button_0 (volatile int *) (CR_MEM_BASE + 0x1c)
#define CR_Button_1 (volatile int *) (CR_MEM_BASE + 0x20)
#define CR_Switch   (volatile int *) (CR_MEM_BASE + 0x24)
```

Code Snippet 6 - rvc_defines.h

As you can see, the file contains the memory address of each memory area. In addition, the file contains the address of the control registers. Also, the file contains several basic functions for writing and reading from memory (WRITE_REG & READ_REG).

From here, all that remains to be done to realize HW API is to write dedicated programs that interface with this file and define dedicated functions for certain purposes. We wrote two APIs in two different domains. The first area is communication between the software and the peripheral components that are on the FPGA and the second was communication between the software and the screen. In Code Snippet 7 you can see the functions signature of the Graphic Screen API and in Code Snippet 8 you can see the functions signature of the CR API.

```
/* Graphic Screen API functions */
void draw_char(char note, int raw, int col) /* This function print a char note on the screen in
(raw,col) position */

void rvc_printf(const char *c) /* This function print a string on the screen in
(CR_CURSOR_V,CR_CURSOR_H) position */

void draw_symbol(int symbol, int raw, int col) /* This function print a symbol from anime table on the
screen in (raw,col) position */

void clear_screen() /* This function clear the screen */

void set_cursor(int raw, int col) /* This set the cursor in (raw,col) position */
void draw_horizontal_line(short int start, short int end, short int raw) /* This function draws
horizontal line from start to and in raw */
void clear_horizontal_line(short int start, short int end, short int raw) /* This function clears
horizontal line from start to and in raw */
void draw_vertical_line(short int start, short int end, short int col) /* This function draws vertical
line from start to and in col */
void delay(int num) /* this function makes delay */
```

Code Snippet 7 - graphic_screen_api

```
/* Control registers API functions */
/* Write functions */
void cr_seg7_0_write(unsigned int val);
void cr_seg7_1_write(unsigned int val);
void cr_seg7_2_write(unsigned int val);
void cr_seg7_3_write(unsigned int val);
void cr_seg7_4_write(unsigned int val);
void cr_seg7_5_write(unsigned int val);
void cr_led_write(unsigned int val);

/* Read functions */
int cr_seg7_0_read();
int cr_seg7_1_read();
int cr_seg7_2_read();
int cr_seg7_3_read();
int cr_seg7_4_read();
int cr_seg7_5_read();
int cr_led_read();
int cr_button_0_read();
int cr_button_1_read();
int cr_switch_read();
```

Code Snippet 8 - cr_api

In addition to this, we have defined an ASCII table and a symbol table which are loaded into the program memory. The tables were defined in the *graphic_screen.h* file, so that any program that wishes to use those tables can interface with this file. We defined the ASCII table in a smart way. First, so that the characters can be distinguished well, we defined two tables ASCII_TOP and ASCII_BOTTOM. As can be seen in Figure 15, the character 'A' consists of a total of 64 bits. The TOP part of the character consists of the top 32 bites – from byte 0 to byte 3, and the BOTTOM part of the character consists of the bottom 32 bits – from byte 4 to byte 7. Therefore, to write the character 'A' we need to turn on the bits marked in black in the drawing. That is, the bits that are marked in black will have the value 1 and all the other bits will have the value 0. In addition to this, to write the letter 'A' to the screen, we will have to write the TOP part of the letter and with an offset of 320 bytes (because there are 80 bytes in each line, and we want to reach the appropriate byte but 4 lines below) we will write the BOTTOM part of the letter.

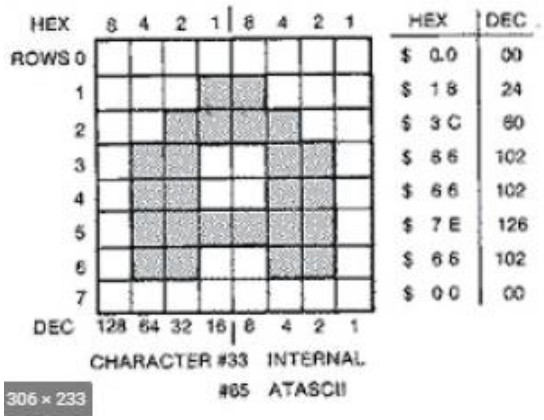

Figure 15 - A character bits drawing

To create the characters and symbols we used a tool called Stella-Graph. Stella-Graph is a program that help to create playfield graphics for the Atari 2600. It allows to create graphics by clicking on the check boxes and then hit the create code button. The program generates the appropriate hex values to include in the source code. This tool provided us with the coding of each letter and each symbol so that we created pre-coded defines for the characters and symbols that are ready to use.

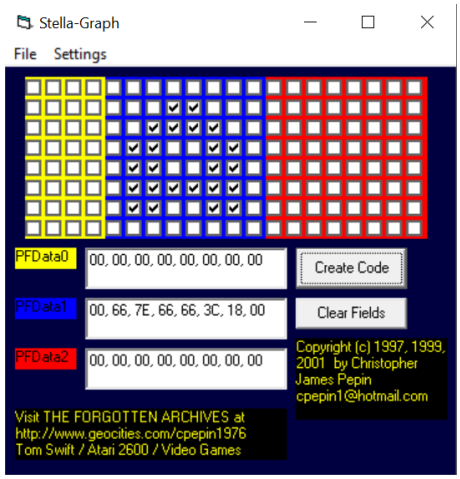

Figure 16 - A character coded in Stella-Graph

The coding of the letter 'A' in the software is:

```
#define A_TOP          0x663C1800
#define A_BOTTOM       0x00667E66
```

Code Snippet 9 - A character coded in the software

This technique was applied to every character and symbol we wanted, and we created the ASCII table in such a way that to get the bitmap of a certain character we turn to the array in the place that corresponds to the ASCII value of this character. For example, the ASCII value of the character 'A' is 65, therefore in place of 65 in the ASCII_TOP table the top part of the letter 'A' appears and in the ASCII_BOTTOM table in place 65 the bottom part of the letter 'A' appears as we defined it beforehand.

```
/* ASCII tabels */
unsigned int ASCII_TOP[97] =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,SPACE_TOP,
EXCL_TOP,0,0,0,0,0,0,0,0,0,0,COMMA_TOP,0,POINT_TOP,0,ZERO_TOP,ONE_TOP,TWO_TOP,THREE_TOP,FOUR_TOP,FIVE_
TOP,SIX_TOP,SEVEN_TOP,EIGHT_TOP,NINE_TOP,COLON_TOP,0,0,0,0,0,0,A_TOP,B_TOP,C_TOP,D_TOP,E_TOP,F_TOP,G_T
OP,H_TOP,I_TOP,J_TOP,K_TOP,L_TOP,M_TOP,N_TOP,O_TOP,P_TOP,Q_TOP,R_TOP,S_TOP,T_TOP,U_TOP,V_TOP,W_TOP,X_T
OP,Y_TOP,Z_TOP};
unsigned int ASCII_BOTTOM[97] =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,SPACE_BOTTOM,EXCL_BOTTOM,0,0,0,0,0,0,
0,0,0,0,COMMA_BOTTOM,0,POINT_BOTTOM,0,ZERO_BOTTOM,
ONE_BOTTOM,TWO_BOTTOM,THREE_BOTTOM,FOUR_BOTTOM,FIVE_BOTTOM,SIX_BOTTOM,
SEVEN_BOTTOM,EIGHT_BOTTOM,NINE_BOTTOM,COLON_BOTTOM,0,0,0,0,0,0,A_BOTTOM,
B_BOTTOM,C_BOTTOM,D_BOTTOM,E_BOTTOM,F_BOTTOM,G_BOTTOM,H_BOTTOM,I_BOTTOM,
J_BOTTOM,K_BOTTOM,L_BOTTOM,M_BOTTOM,N_BOTTOM,O_BOTTOM,P_BOTTOM,Q_BOTTOM,R_BOTTOM,S_BOTTOM,T_BOTTOM,U_B
OTTOM,V_BOTTOM,W_BOTTOM,X_BOTTOM,Y_BOTTOM,Z_BOTTOM};
```

Code Snippet 10 - ASCII table

6.4 SW GUIDE

In this section we will go over the installations and steps required to become a contributor to the project. This is a technical guide detailing the steps required to do so.

1) Download a text editor:
   - VSCode: https://code.visualstudio.com/download
   - Add useful extensions – Vim, PowerShell, System-Verilog, Venus Terminal.

2) Download git-bash for windows:
   - https://gitforwindows.org/

3) Set git-bash in VSCode:
   - You may configure the ~/.bashrc & ~/.aliases with your preferences.

4) Download ModelSim – A System-Verilog compiler & simulator (lite free version):
   - https://www.intel.com/content/www/us/en/software-kit/660907/intel-quartus-prime-lite-edition-design-software-version-20-1-1-for-windows.html
     Download individual files: ModelSim, Quartus, Intel MAX 10 Device Support.
     Note: after all 3 programs ere downloaded, run the Quartus installation which will automatically install ModelSim & MAX10.

5) Download RISC-V ToolChain:
   - https://github.com/xpack-dev-tools/riscv-none-embed-gcc-xpack/releases/ - File name: *xpack-riscv-none-embed-gcc-10.1.0-1.1-win32-x64.zip*.
   - https://xpack.github.io/riscv-none-embed-gcc/install/ - Follow manual install (only extract in correct location).

6) git-bash shell – Set aliases for the compile & link commands:

   define the following aliases in *~/.aliases* or *C:\ProgramFiles\Git\etc\profile.d\aliases.sh*:
   - *alias rv_gcc='/c/Users/YoursUserName/AppData/Roaming/xPacks/riscv-none-embed-gcc/xpack-riscv-none-embed-gcc-10.1.0-1.1/bin/riscv-none-embed-gcc.exe'*
   - *alias rv_objcopy='/c/Users/ YoursUserName /AppData/Roaming/xPacks/riscv-none-embed-gcc/xpack-riscv-none-embed-gcc-10.1.0-1.1/bin/riscv-none-embed-objcopy.exe'*
   - *alias rv_objdump='/c/Users/ YoursUserName/AppData/Roaming/xPacks/riscv-none-embed-gcc/xpack-riscv-none-embed-gcc-10.1.0-1.1/bin/riscv-none-embed-objdump.exe'*

   This step is just for convenience in order not to use long commands.

7) Clone the repository:
   - *git clone https://github.com/amichai-bd/rvc_asap.git*

8) Create a new branch:
    - *git checkout -b "branch_name"*

9) Create a simple C program under *apps/C* directory – *alive.c*:

```c
#include "../defines/rvc_defines.h"

int main(){
    volatile int *ptr = (int*) D_MEM_BASE;
    int x,y,z;
    x = 2;
    y = 3;
    z = x+y;
    ptr[0] = z;

    return 0;
}
```

10) Run from git-bash terminal in *buildl.sh* script directory the following command:
    - *./build.sh alive*
    - For GUI interface simulation add the *-gui* flag and for debug add *-debug* flag (don't delete temporary compilation files).

11) Check you got the result 5 in address 0x00004000 in *target/alive/mem_snapshot.log* file:

```
Offset 00004000 : 00000005
Offset 00004004 : xxxxxxxx
Offset 00004008 : xxxxxxxx
```

12) Modifies, staging and commits:
    - *git add . git commit -m 'your commit'*

13) Pull from origin/master (to make sure no conflicts):
    - *git pull origin main*

14) Push to origin:
    - *git push origin 'branch_name'*

15) Modifies, staging and commits:
    - *git add . git commit -m 'your commit'*

16) Add a pull request:
    - From https://github.com/amichai-bd/rvc_asap/pulls ->New pull request

6.5 WORKING EXAMPLE

When planning software in a low-resource system (in our case, without a doubt, the system is low in resources) one of the most important aspects to consider is the memory limit. The code you write should be smart and efficient code that takes this limitation into account. This is reflected in the types of variables that are worked with, in the code summaries so that no duplicate code is created and in the number of constitutive calls to the functions so as not to overload the stack. Constitutive calls to functions can overload the Stack because the function's arguments, return address, and local variables of the function enter the Stack. Many constitutive calls to a function can create Stack Overflow. So, let's start from the end, we managed to write the old and classic Breakout Game which was a very popular game in the early 80's with the memory limitations we had.

The way we were able to do this is by being careful and paying attention to the principles we have mentioned so far. We tried to use types like *short int* whose size is 2 bytes instead of *int* whose size is 4 bytes, we tried to avoid repetition of double code and many constitutive calls to functions in order not to overload the Stack. For it to be possible to play the game in a comfortable way that can be viewed by the human eye, we had to insert delays in the software since the screen refresh rate is 60 times per second.

The game is structured so that there is a game brick that exists at the bottom of the screen and a ball that moves in the space of the screen and hits the brick board that is on the screen. The player must move the game brick so that the ball does not fall to the ground, when the ball hits the game brick it goes back up and receives a certain direction of movement. The game ends when the player fails to catch the ball. 3 things happen at each stage increase, first the game brick gets shorter. Second, the speed of the ball increases and the last is that the brick wall rolls down one step, and a new row of bricks enters. Each brick that is blown up by the ball gives the player one point. The player can move the game board using the Key[0] and Key[1] buttons of the FPGA.

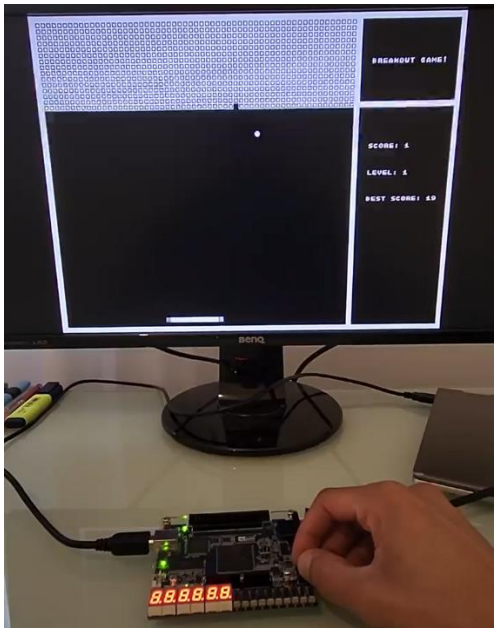

Figure 17 - Breakout game on RVC_ASAP

7 TFM: TOOLS, FLOW & METHODOLOGY

7.1 TOOLS

Git Bash

Git Bash is an application for Microsoft Windows environments that provides an emulation layer for the Git command line experience. Bash stands for Bourne Again Shell. A shell is a terminal application used to interface with an operating system using scripted commands. Bash is a popular default shell on Linux and macOS. Git Bash is a package that installs Bash, some common Bash and Git utilities on a Windows operating system.

ModelSim

ModelSim is a multi-language environment for simulation of hardware description languages such as VHDL, Verilog and SystemC. Simulation is performed using the graphical user interface (GUI), or automatically using scripts. On the RVC_ASAP project we used ModelSim in the validation process to simulate the RVC_ASAP core and the memories using the System Verilog RTL files. On the first steps of the project, we used the GUI form of ModelSim, especially in the Waveform GUI to debug and track signals, but as the project progressed, we used automation scripting to simulate.

Quartus

Intel Quartus is programmable logic device design software. Quartus Prime enables analysis and synthesis of HDL designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. Quartus Prime includes an implementation of VHDL and Verilog for hardware description, visual editing of logic circuits, and vector waveform simulation.

Visual Studio Code

Visual Studio Code is a streamlined code editor with support for development operations like debugging, task running, and version control. It aims to provide just the tools a developer needs for a quick code-build-debug cycle and leaves more complex workflows to fuller featured IDEs, such as Visual Studio IDE.

It was very convenient for us to manage the project in this environment, mainly because you can add extensions that support Git Bash or the Venus simulator for example.

7.2 BUILD

The main flow of the build process shown in Figure 18.

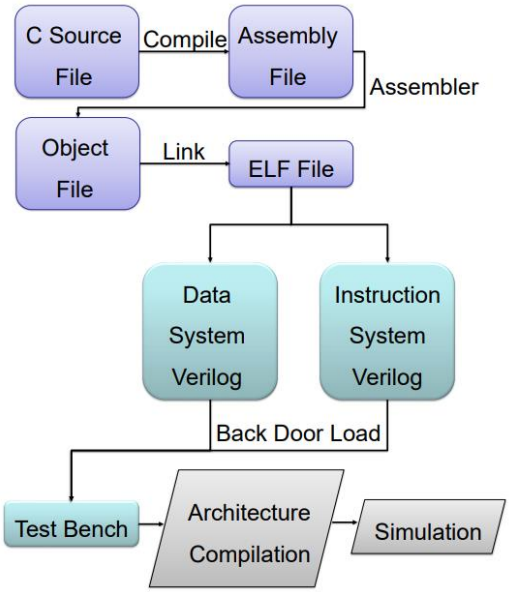

Figure 18 - Build process flow

We built a main build script called buildl.sh which is written in the Bash language and performs the entire process described in Figure 18 automatically. We expand on the script itself here, but we will now expand on the core of the script and its building blocks.

First, we defined the following aliases in *~/.aliases* or *C:\ProgramFiles\Git\etc\profile.d\aliases.sh*:

- *alias rv_gcc='/c/Users/YoursUserName/AppData/Roaming/xPacks/riscv-none-embed-gcc/xpack-riscv-none-embed-gcc-10.1.0-1.1/bin/riscv-none-embed-gcc.exe'*
- *alias rv_objcopy='/c/Users/ YoursUserName /AppData/Roaming/xPacks/riscv-none-embed-gcc/xpack-riscv-none-embed-gcc-10.1.0-1.1/bin/riscv-none-embed-objcopy.exe'*
- *alias rv_objdump='/c/Users/ YoursUserName/AppData/Roaming/xPacks/riscv-none-embed-gcc/xpack-riscv-none-embed-gcc-10.1.0-1.1/bin/riscv-none-embed-objdump.exe'*

This step is just for convenience in order not to use long commands.

Now, we will go through the various steps in the build process and indicate how they are realized. We will run throw a live example for the build process. At each stage, we will touch on the important details for that stage and attach figures that illustrate what is being done at each stage.

In total, we built 24 tests in C and Assembly languages that test different aspects of the processor. Let's take, for example, a basic test that performs Bubble Sort. The test is called *BubbleSort.c* and it performs Bubble Sort on integers array.

```c
#define _ASMLANGUAGE
#include "../defines/rvc_defines.h"
void swap(int *xp, int *yp){
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n-1; i++)
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}
int main(){
    int arr[] = {80,200,60,300,100,70,90};
    bubbleSort(arr,7);

    for(int i=0;i<7;i++)        MEM_SCRATCH_PAD[i] = arr[i];
    return 0;
}
```

Code Snippet 11 - BubbleSort.c

7.2.1 SW Build

1. **Compile** – Compile the C program to RV32I Assembly code by the following command:

   rv_gcc -S -ffreestanding -march=rv32i BubbleSort.c -o BubbleSort.s

```
main:
    addi    sp,sp,-48
    sw  ra,44(sp)
    sw  s0,40(sp)
    addi    s0,sp,48
    lui a5,%hi(.LC0)
    addi    a5,a5,%lo(.LC0)
    lw  a6,0(a5)
    lw  a0,4(a5)
    lw  a1,8(a5)
    lw  a2,12(a5)
    lw  a3,16(a5)
    lw  a4,20(a5)
    lw  a5,24(a5)
    sw  a6,-48(s0)
    sw  a0,-44(s0)
    sw  a1,-40(s0)
    sw  a2,-36(s0)
    sw  a3,-32(s0)
    sw  a4,-28(s0)
```

Code Snippet 12 - BubbleSort.s

Since the total output is large, we only show the part from the main function as it is shown in Assembly language.

2. **Assembler & Linker** – Assemble the Assembly program to object file and link the output with the assembled *crt0.s* file – *crt0.o* to get ELF file (Executable and linkable Format) by the following command:

   rv_gcc -O3 -march=rv32i -Tlink.common.ld -nostartfiles -D__riscv__ crt0.s BubbleSort.s -o BubbleSort.elf

```
000001dc <main>:
 1dc:   fd010113            addi    x2,x2,-48
 1e0:   02112623            sw  x1,44(x2)
 1e4:   02812423            sw  x8,40(x2)
 1e8:   03010413            addi    x8,x2,48
 1ec:   000047b7            lui x15,0x4
 1f0:   00078793            mv  x15,x15
 1f4:   0007a803            lw  x16,0(x15) # 4000 <_endtext+0x3d64>
 1f8:   0047a503            lw  x10,4(x15)
 1fc:   0087a583            lw  x11,8(x15)
 200:   00c7a603            lw  x12,12(x15)
 204:   0107a683            lw  x13,16(x15)
 208:   0147a703            lw  x14,20(x15)
```

Code Snippet 13 - BubbleSort.elf

It can be noticed that the names of the processor registers have been changed, the reason for this lies in the fact that in the previous step the program was presented in disassembly language, and now this is a complete Assembly presentation according to RV32I ISA. For example, SP (Stack Pointer) in RV32I ISA is x2 register.

3. **ELF to SV** – Compile the ELF file to System Verilog file (*.sv*) which contains the instruction and data memories content. buildl.sh script will separate this single output file into two separate files so that each one is loaded separately in Test Bench to its appropriate memory region (Instruction Memory or Data Memory) by the following command:

   rv_objcopy --srec-len 1 --output-target=verilog BubbleSort.elf BubbleSort-inst_mem_rv32i.sv

```
@00004000
50 00 00 00 C8 00 00 00 3C 00 00 00 2C 01 00 00
64 00 00 00 46 00 00 00 5A 00 00 00
```

Code Snippet 14 - BubbleSort_Data.sv

```
@00000000
13 00 00 00 13 00 00 00 13 00 00 00 13 00 00 00
13 00 00 00 6F 00 40 00 17 71 00 00 13 01 81 FE
B7 7F 00 00 93 8F 8F 02 93 00 00 00 23 A0 1F 00
23 A2 1F 00 93 81 00 00 13 82 00 00 93 82 00 00
13 83 00 00 93 83 00 00 13 84 00 00 93 84 00 00
13 85 00 00 93 85 00 00 13 86 00 00 93 86 00 00
13 87 00 00 93 87 00 00 13 88 00 00 93 88 00 00
13 89 00 00 93 89 00 00 13 8A 00 00 93 8A 00 00
13 8B 00 00 93 8B 00 00 13 8C 00 00 93 8C 00 00
13 8D 00 00 93 8D 00 00 13 8E 00 00 93 8E 00 00
13 8F 00 00 93 8F 00 00 EF 00 40 13 73 00 10 00
```

Code Snippet 15 - BubbleSort_Instruction.sv

It can be noticed that the array we defined is loaded into the data region at address 0x00004000 as defined in the linker. For example, the first member of the array is 80 in decimal base which is 50 in hexadecimal base as you can see. Moreover, it can be noticed that the first instruction found in the instruction memory at address 0x00000000 as defined in the linker is 00000013 which is a NOP instruction as defined in the *crt0.s* file.

7.2.2 HW Build

1. **Hardware Compilation** – Compile all the architecture including the Test Bench and all the files present in the *list.f* file by the following command:

   vlog.exe +define+HPATH= BubbleSort -f list.f

```
//include dir - the Machine code for I_MEM load.
+incdir+../verif/
+incdir+../source/common
+define+SIMULATION_ON

//Source File (PKG, RTL, MACROS)
../source/common/rvc_asap_pkg.sv
../source/rvc_asap_5pl/rvc_asap_5pl.sv
../source/rvc_asap_5pl/rvc_asap_5pl_mem_wrap.sv
../source/rvc_asap_5pl/rvc_asap_5pl_i_mem.sv
../source/rvc_asap_5pl/rvc_asap_5pl_d_mem.sv
../source/rvc_asap_5pl/rvc_asap_5pl_cr_mem.sv
../source/rvc_asap_5pl/rvc_asap_5pl_vga_mem.sv
../source/rvc_asap_5pl/rvc_asap_5pl_vga_ctrl.sv
../source/rvc_asap_5pl/rvc_asap_5pl_sync_gen.sv
../source/rvc_asap_5pl/rvc_top_5pl.sv

//Test Bench
../verif/rvc_asap_5pl_tb.sv
```

Code Snippet 16 - list.f file

2. **Simulation** – Simulate the Test Bench with ModelSim by the following command:

   vsim.exe -gui work.tb

   To debug the hardware, many times we had to open a wave diagram of the control signals by using ModelSim software. The debug technique was to follow the signals in relation to what we would expect to see and what we see. If we detected an anomaly, then we returned to the implementation of the hardware to understand what caused the anomaly and thus correct it. The key to success in these cases is to start by building simple test plans and gradually increase the complexity of the plans. A wave diagram of running the *BubbleSort.c* program on the design is given as an example in Figure 19.

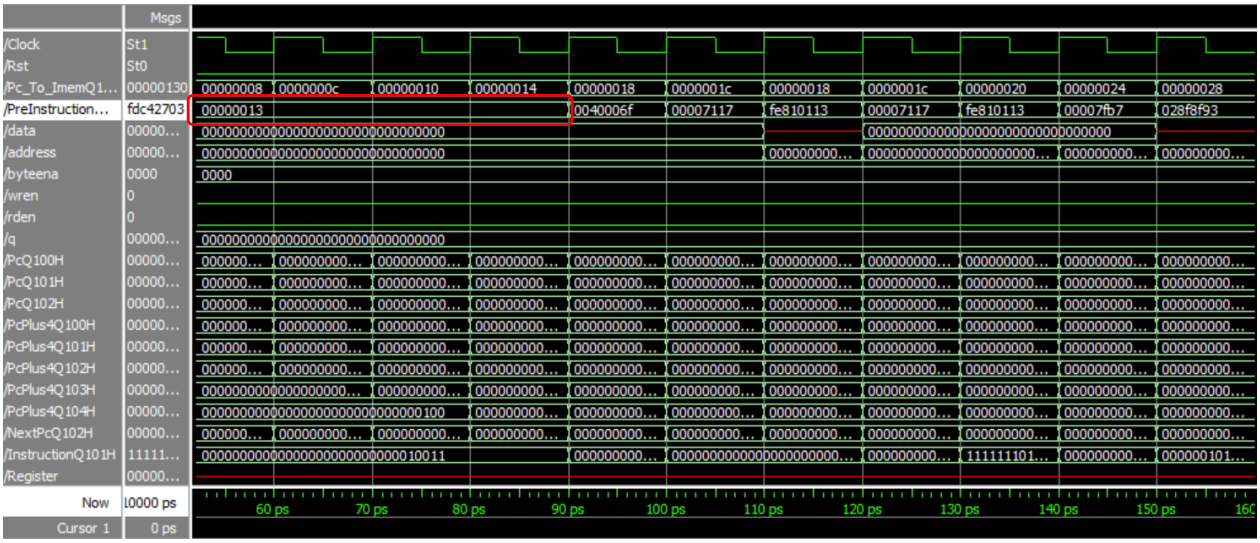

Figure 19 - BubbleSort.c Wave Diagram

In the attached wave diagram, you can see the NOP instruction that appears in the *crt0.s* file in the square surrounded by red. Of course, during the past year we encountered several complex bugs that required deep and gentle treatment and the wave diagram tool was a necessary and important tool for solving them.

3. **Check the results** – In the final stage we check the results by conduct a comparison between actual output of the test to the memory and its golden image as described in Checker section.

```
Offset 00004000 : 0000003c
Offset 00004004 : 00000046
Offset 00004008 : 00000050
Offset 0000400c : 0000005a
Offset 00004010 : 00000064
Offset 00004014 : 000000c8
Offset 00004018 : 0000012c
```

Figure 20 - BubbleSort.c golden image

```
Offset 00004000 : 0000003c
Offset 00004004 : 00000046
Offset 00004008 : 00000050
Offset 0000400c : 0000005a
Offset 00004010 : 00000064
Offset 00004014 : 000000c8
Offset 00004018 : 0000012c
```

Figure 21 - BubbleSort.c real image

As you can see, the snapshots match and the numbers in the array are sorted as required. As a result, the test was passed successfully.

7.3 AUTOMATION & SCRIPTS

7.3.1 buildl.sh

buildl.sh is the main build script. The idea behind the development of the script were to enable a convenient and simple way to perform the validation and simulation process, for many test programs. The script was written in the Bash language, which is a common scripting language, convenient and simple to use. The script contains the following steps:

1. Check build flags.
2. Print usage in case there are no arguments provided by the user.
3. Check if the directories *apps/elf*, *apps/elf_txt*, *apps/sv* exist – if not create them.
4. Clean *target* directory and compilation directories.
5. Compile C files in *apps/C* directory to Assembly files in *apps/asm* directory.
6. Compile Assembly files in *apps/asm* directory to ELF files in *apps/elf* directory.
7. Compile ELF files in *apps/elf* directory to SV files in *apps/sv* directory.
8. Split SV files to Instruction Memory and Data Memory and call to hex2mif.py script.
9. Compile the design with *vlog.exe*.
10. Simulate the test with ModelSim (*vsim.exe*).
11. Clean compilation files.
12. Check the results.
13. Print summary.

Figure 22 illustrates the usage of the script in case no arguments provided.

```
$ ./buildl.sh
usage: ./buildl.sh [arch] [gui] [test_0] [test_1] ... [test_n]
  all                           Build all tests
  All                           Build all tests
  arch                          [-sc] [-5pl]
                                Basic arch is 5pl
  gui                           [-gui]
  compile                       [-compile]
  just asm                      [-asm_verif]
  library examples              [-lib]
  don't clean compilation files [-debug]
  example_1 : ./build.sh all
  example_2 : ./build.sh ALL
  example_3 : ./build.sh basic_commands1 basic_commands2 ... basic_commandsn
  example_4 : ./build.sh -sc -gui basic_commands1 basic_commands2 ... basic_commandsn
  example_5 : ./build.sh -5pl -gui basic_commands1 basic_commands2 ... basic_commandsn
  example_5 : ./build.sh -5pl basic_commands1 basic_commands2 ... basic_commandsn
```

Figure 22 - buildl.sh script usage

Figure 23 illustrates correct running output of the script for *test_asap.c* test.

```
=========================================================
=========================================================
=========================================================
              Check: test_asap
=========================================================
              The test ended successfully!
                        😃
=========================================================
              End Check: test_asap
=========================================================
=========================================================
====================== Summary ==========================
Total tests  : 1
Tests Pass   : 1
Tests Fail   : 0
Tests unknown: 0
=========================================================
```

Figure 23 - buildl.sh script - correct running output

7.3.2 hex2mif.py

A *.mif* file is an ASCII text file that specifies the initial content of a memory block (RAM or ROM), that is, the initial values for each address. This file is used during project compilation and/or simulation. You can create a Memory Initialization File in the Memory Editor, the In-System Memory Content Editor, or the Quartus Prime Text Editor. You can also use a hexadecimal (Intel-Format) file (.hex) to provide memory initialization data. A Memory Initialization File contains the initial values for each address in the memory. More details about if can found in this link.

In our case, our output from the build process was a *.hex* text file so we need to convert it to *.mif* file. Actually, on Linux system or WSL, there is a command that generate it - *srec_cat -o –mif* but it has not been possible in our case and it can be helpful to implement something like this by hand.

The main idea of this script is quite simple, to convert System-Verilog fie (*.sv*):

```
@00004000
C0 00 00 00 F9 00 00 00 A4 00 00 00 B0 00 00 00
99 00 00 00 92 00 00 00 82 00 00 00 F8 00 00 00
80 00 00 00 90 00 00 00
```

Figure 24 - System Verilog file

To it appropriate *.mif* file:

```
-- SUPPOSE! be similar to->
-- Generated automatically by srec_cat -o --mif
--
DEPTH=268;
WIDTH=32;
ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;
CONTENT BEGIN;
0000: 00000013 00000013 00000013 00000013 00000013 0040006F 00007117 FE810113;
0008: 00007FB7 028F8F93 00000093 001FA023 001FA223 00008193 00008213 00008293;
0010: 00008313 00008393 00008413 00008493 00008513 00008593 00008613 00008693;
0018: 00008713 00008793 00008813 00008893 00008913 00008993 00008A13 00008A93;
0020: 00008B13 00008B93 00008C13 00008C93 00008D13 00008D93 00008E13 00008E93;
0028: 00008F13 00008F93 0C0000EF 00100073;
```

Figure 25 - MIF file

we see that in the *.sv* file there are bulks of bytes (8-bit - 2 hex digit) and that each line is end with size of 4 words (32 bit × 4) or less. Moreover, that some line starts with *@0xxxx* that represent the address it points. In the *.mif* file, there are bulks of words (32-bit - 8 hex digit), and each line start with number represent the number of words in each line cumulatively, and the line ends when reached to 8 word, or when reached other/far address (*@0xxx* in the *.sv* file).

So, what we've done is gather all the bytes into words (take care of the Little-Endian format) and start each line by the cumulative number of words, or by the parallel amount in case there is address that far in the *.sv* file.

7.4 System-Verilog Coding Style

Throw the entire System Verilog programming of the RTL, we used a coding style for order, uniformity, and readability.

- **Macros** - On system Verilog, when you want do design a flip-flop it is known to use this convention:

```
`define  RVC_EN_RST_MSFF(q,i,clk,en,rst)\
         always_ff @(posedge clk)       \
            if (rst)    q <='0;         \
            else if(en) q <= i;
```

Code Snippet 17 - Flip-Flop in System Verilog

Therefore, every time we had to use a flip-flop, we used the define which was defined in advance.

- **CamelCase** - we used CamelCase coding style convention to name the variables, logic signals, wires, buffers etc. throw all the RTL files. On this convention we name the variables as follows: SomeNameOfTheLogic with no spaces with each word starts, for example: SimpleReset.

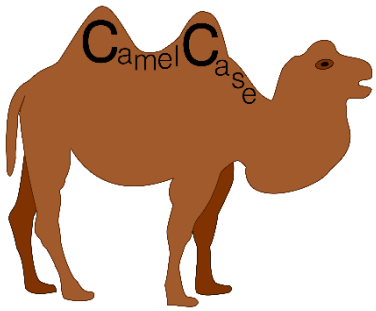

Figure 26 - CamelCase coding style

- **rvc_asap_pkg** – All the RVC_ASAP project parameters were defined in a file named rvc_asap_pkg.sv. Many parameters such as opcodes, CR addresses, memory sizes, offsets, MSB and LSB locations, encoded region (I_MEM, D_MEM, CR_MEM, VGA_MEM) bits, structs (like CR memory struct) and more were defined there and can be used on all RTL files.

```
typedef enum logic [6:0] {
   LUI    = 7'b0110111 ,
   AUIPC  = 7'b0010111 ,
   JAL    = 7'b1101111 ,
   JALR   = 7'b1100111 ,
   BRANCH = 7'b1100011 ,
   LOAD   = 7'b0000011 ,
   STORE  = 7'b0100011 ,
   I_OP   = 7'b0010011 ,
   R_OP   = 7'b0110011 ,
   FENCE  = 7'b0001111 ,
   SYSCAL = 7'b1110011
} t_opcode ;
```

Code Snippet 18 - RV32I instructions opcode parameters from rvc_asap_pkg.sv

- **Cycle Suffix** – To make development easier, we labeled each signal name with a suffix implying the pipeline stage this signal is used, generated, or sampled. After the signal sampled in flops, its suffix increased by 1. Any signal can be used anywhere on the cycle but that is our convention. For example, AluOutQ102H is the ALU output signal on stage Q102H and when it used in stage Q103H it will be sampled on flop and will be named AluOutQ103H.

```
// Q102H to Q103H Flip Flops
`RVC_MSFF(RegRdData2Q103H     , RegRdData2Q102H     , Clock)
`RVC_MSFF(AluOutQ103H         , AluOutQ102H         , Clock)
`RVC_MSFF(CtrlDMemByteEnQ103H , CtrlDMemByteEnQ102H , Clock)
`RVC_MSFF(CtrlDMemWrEnQ103H   , CtrlDMemWrEnQ102H   , Clock)
`RVC_MSFF(SelDMemWbQ103H      , SelDMemWbQ102H      , Clock)
`RVC_MSFF(CtrlSignExtQ103H    , CtrlSignExtQ102H    , Clock)
`RVC_MSFF(PcPlus4Q103H        , PcPlus4Q102H        , Clock)
`RVC_MSFF(SelRegWrPcQ103H     , SelRegWrPcQ102H     , Clock)
`RVC_MSFF(RegDstQ103H         , RegDstQ102H         , Clock)
`RVC_MSFF(CtrlRegWrEnQ103H    , CtrlRegWrEnQ102H    , Clock)
`RVC_MSFF(flushQ103H          , flushQ102H          , Clock)
```

Code Snippet 19 - Signal from Q102H stage, sampled before used in Q103H stage

- **Files Headers** – In all the project files, we used a uniform template for the headers of the project files. The goal was to maintain uniformity and provide an informative and clear header for each file in the project.

```
//------------------------------------------------------------------------------
// Title            : riscv as-fast-as-possible
// Project          : rvc_asap
//------------------------------------------------------------------------------
// File             : rvc_asap_5pl
// Original Author  : Amichai Ben-David
// Code Owner       :
// Adviser          : Amichai Ben-David
// Created          : 10/2021
//------------------------------------------------------------------------------
// Description :
// This module will comtain a complite RISCV Core supportint the RV32I
// Will be implemented in a single cycle microarchitecture.
// The I_MEM & D_MEM will support async memory read. (This will allow the single-cycle arch)
// ---- 5 Pipeline Stages -----
// 1) Q100H Instruction Fetch
// 2) Q101H Instruction Decode
// 3) Q102H Execute
// 4) Q103H Memory Access
// 5) Q104H Write back data from Memory/ALU to Registerfile
```

Code Snippet 20 - RVC_ASAP file header

7.5 GIT & GITHUB

7.5.1 Git

Git is a well-known software for tracking changes in any set of files. we used git to clone our project files to our personal computers, work on them simultaneously and then upload them to our common cloud stored in *github.com*. Git also used us for solving merge conflicts and track changes.

7.5.2 GitHub

GitHub is a free provider of internet hosting for software development and version control using Git. We used GitHub to store our repository which include all the project files.

The structure of our project repository is as follows:

```
|    buildl.sh
|    .gitignore
|    LICENSE
|    README.md
+---apps
|       +---asm
|       +---asm_verif
|       +---C
|       +---elf
|       +---elf.txt
|       +---library
|       +---sv
|       crt0.s
|       link.common.ld
|       README.md
+---doc
|       HOW_TO.md
|       5 stage pipeline.pptx
|       HOW_TO_RUN_BUILDL.md
|       HOW_TO_MAKE_LINKER.md
|       HOW_TO_GIT.md
|       riscv-spec-20191213.pdf
|       rvc_asap.pptx
|       rvc_core_asap.vsdx
|       single_cycle_RVC.png
```

```
+---FPGA
|       +---db
|       +---incremental_db
|       +---mem_hex
|       +---output_files
|       +---sv
|       assignment_defaults.qdf
|       CPU_GARAGE.qsf
|       platform.sv
|       rvc_asap.qpf
+---modelsim_run
|       README.md
|       rvc_asap_5pl_list.f
|       rvc_asap_sc_list.f
|       +---work
+---source
|       README.md
|       +---common
|       +---rvc_asap_5pl
|       +---rvc_asap_sc
+---target
Temporary compilation files.
+---verif
|       README.md
|       rvc_asap__5pl_tb.sv
|       +---golden_image
```

We will go through the different folders in the repository and explain the essence of each folder. *apps* directory consists of all the software content. Means all the Assembly programs and the C programs are in this folder. The directories: *C*, *asm_verif*, and *library* contains static codes. The directories: *asm*, *elf*, *elf.txt*, and *sv* contains temporary compilation files which are cleaned by buildl.sh script. *source* directory contains all the RTL files of the design. *doc* directory contains all the documentation of the project. *FPGA* directory contains all the files related to the FPGA. *modelsim_run* contains all the files related to ModelSim simulation. *target* directory contains temporary compilation files such as memory dump which are cleaned by buildl.sh script. *verif* directory contains the Test Bench and golden images of the various tests.

Using the *.gitignore* file we prevent irrelevant files from being uploaded to the repository.

```
# Ignore those extensions
*.qdb
*.qpg
*.qtl
*.bak
_info
# modelsim generated files
*.swp
modelsim/work/*
modelsim_run/*
target/*
transcript
```

Figure 27 - .gitignore file

In addition to storing our files and tracking old version using git, the GitHub also include many aspects:

- **Code & Code Review** - Each contributor can upload his code and receive code review before the new uploaded code is merged with the complete repository code.

Figure 28 - RVC_ASAP repository structure

- **Task Management** - Used by Issues tab on GitHub. Each meeting with the advisor we assign tasks between us and detailed about the task in its page.

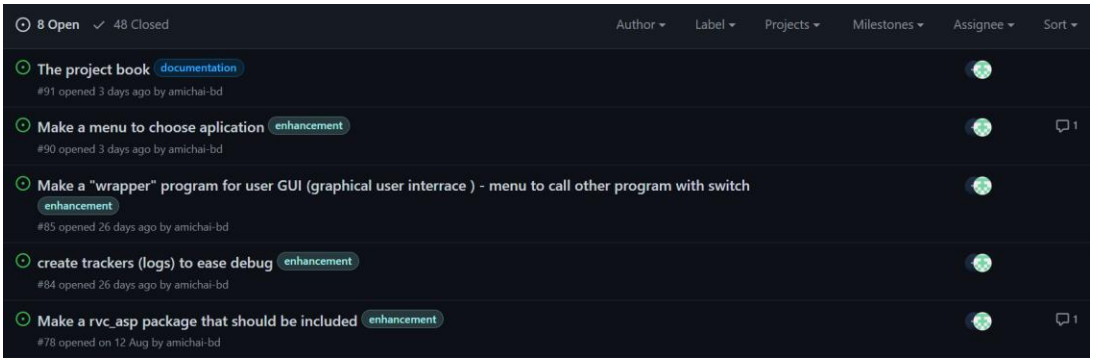

Figure 29 - Project tasks as shown in GitHub

- **Discussions and Questions** - Every time we encountered a problem in our work, we wrote about it in the Discussions tab. One of the many contributors to one of the projects can answer if knows.

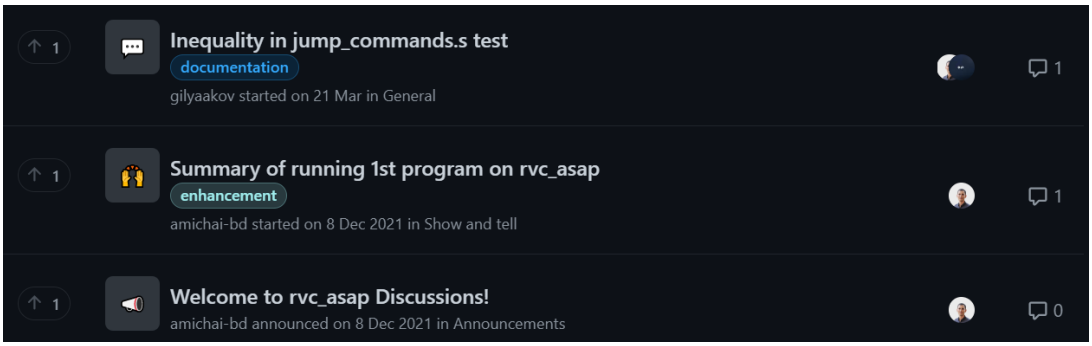

Figure 30 - Questions & Discussions on GitHub repository

8 FUTURE PLANS

We think this project is an amazing foundation for a wide variety of interesting and fascinating projects in both hardware and software fields. The directions for the continuation concern making the project a system as similar as possible to a standard computer that we know. Each of the directions listed is a basis for a complete graduation project. Here are the plans to continue:

- Expanding the API functions and writing additional service functions such as malloc using a *heap* structure.
- Writing a lite operating system for creating and managing processes including scheduling, control, and interrupt mechanisms.
- Create L1 cache, and a controller for memory management including the implementation of error correction codes.
- Building arithmetic circuits for fast addition, subtraction, multiplication, and division.
- Create OOO (Out-Of-Order) execution by implementing a ROB (Re-Order Buffer).
- Implement Branch Predictor.
- Implement RISC-V extensions (M/F/A/CSR).
- Implement Interrupts & Exceptions.
- IO – UART.
- Connect Keyboard.
- RGB Mode – lower the resolution of the screen but get RGB capabilities.

9 REFERENCES

- RISC-V official ISA specifications.
  https://riscv.org/technical/specifications/

- RISC-V official organization's website.
  https://riscv.org/

- Wikipedia, the open online encyclopedia.
  https://en.wikipedia.org/wiki/Main_Page

- Official GNU, about linker scripts.
  https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_3.html

- RISC-V Toolchain install guide and user guide repository by John Winans.
  https://github.com/johnwinans/riscv-toolchain-install-guide

- GNU GCC compiler guide.
  https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

- GitHub – Getting started with Git & GitHub.
  https://docs.github.com/en/get-started/quickstart/hello-world

- Git official, git documentations.
  https://git-scm.com/docs/gittutorial

- System-Verilog tutorial.
  https://verificationguide.com/systemverilog/systemverilog-tutorial/

- ModelSim Edition download and documentation.
  https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html

- VGA Signal Timing.
  http://tinyvga.com/vga-timing

- LOTR project. Adi Levy & Saar Kadosh, Technion
  https://github.com/amichai-bd/riscv-multi-core-lotr