

University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

EECS151/251A - LB, Spring 2021

Project Specification: RISC-V151

Version 1.0

Contents

1	Introduction	4
1.1	Tentative Deadlines	4
1.2	General Project Tips	4
2	Checkpoints 1 & 2 - 3-stage Pipelined RISC-V CPU	5
2.1	Setting up your Code Repository	5
2.2	Integrate Designs from Labs	5
2.3	Project Skeleton Overview	6
2.4	RISC-V 151 ISA	7
2.4.1	CSR Instructions	7
2.5	Pipelining	7
2.6	Hazards	9
2.7	Register File	9
2.8	RAMs	9
2.8.1	Initialization	9
2.8.2	Endianness + Addressing	9
2.8.3	Reading from RAMs	10
2.8.4	Writing to RAMs	10
2.9	Memory Architecture	11
2.9.1	Summary of Memory Access Patterns	11
2.9.2	Unaligned Memory Accesses	11
2.9.3	Address Space Partitioning	12
2.9.4	Memory Mapped I/O	12
2.10	Testing	13
2.11	Riscv151 Tests	14
2.12	Software Toolchain - Writing RISC-V Programs	14
2.13	Assembly Tests	15
2.14	RISC-V ISA Tests	15
2.15	Software Tests: Echo	16
2.16	BIOS and Programming your CPU	17
2.17	Target Clock Frequency	18
2.18	Matrix Multiply	18
2.19	How to Survive This Checkpoint	19
2.19.1	How To Get Started	19

2.20	Checkoff	21
2.20.1	Checkpoint 1: Block Diagram	21
2.21	Questions	21
2.21.1	Checkpoint 2: Base RISCv151 System	22
2.21.2	Checkpoints 1 & 2 Deliverables Summary	23
3	Checkpoint 3 - Hardware-Accelerated Convolutional Neural Network	24
3.1	Checkpoint Overview	24
3.1.1	LeNet	24
3.1.2	New files	25
3.1.3	High-level overview of the full system	26
3.1.4	ARM Baremetal Application	27
3.2	Software Implementation	27
3.2.1	conv3D	27
3.2.2	maxpool2D	28
3.2.3	fconn	29
3.3	Naive conv3D: Sources of Inefficiency	29
3.4	AXI Bus Interface	30
3.4.1	Read Interface Channels	31
3.4.2	Write Interface Channels	31
3.5	Vivado Block Design with Zynq Processing System (PS)	33
3.6	System Integration with Riscv151	34
3.6.1	IO Memory-mapped Addresses for DMA Controller and Accelerator	34
3.6.2	Integrating DMA Controller with CPU Data Memory	34
3.7	LeNet Demo software code	35
3.8	Resources and Suggestions for Accelerator design	35
3.9	High-throughput Spatially-mapped Design (Extra Credit)	35
3.10	Steps to Complete Checkpoint 3	35
3.11	Checkpoint 3 Deliverables	37
4	Final Checkpoint - Optimization	39
5	Grading and Extra Credit	40
5.1	Checkpoints	40
5.2	Style: Organization, Design	40
5.3	Final Project Report	40
5.3.1	Report Details	40
5.4	Extra Credit	41
5.5	Project Grading	41
A	Local Development	43
A.1	Linux	43
A.2	OSX, Windows	43
B	BIOS	43
B.1	Background	43

B.2	Loading the BIOS	44
B.3	Loading Your Own Programs	44
B.4	The BIOS Program	45
B.5	The UART	46
B.6	Command List	46
B.7	Adding Your Own Features	46
C	Debugging with Vivado Integrated Logic Analyzer	47

1 Introduction

The goal of this project is to familiarize EECS151/251A students with the methods and tools of digital design. Working alone or in a team of two, you will design and implement a 3-stage pipelined RISC-V CPU with a UART for tethering. Afterwards, you will build a hardware accelerator (*TBD*) and do a system integration with your RISC-V CPU.

Finally, you will optimize your CPU for performance (maximizing the Iron Law) and cost (FPGA resource utilization).

You will use Verilog to implement this system, targeting the Xilinx PYNQ platform (a PYNQ-Z1 development board with a Zynq 7000-series FPGA). The project will give you experience designing with RTL descriptions, resolving hazards in a simple pipeline, building interfaces, and teach you how to approach system-level optimization.

In tackling these challenges, your first step will be to map the high level specification to a design which can be translated into a hardware implementation. After that, you will produce and debug that implementation. These first steps can take significant time if you have not thought out your design prior to trying implementation.

As in previous semesters, your EECS151/251A project is probably the largest project you have faced so far here at Berkeley. Good time management and good design organization is critical to your success.

1.1 Tentative Deadlines

The following is a brief description of each checkpoint and approximately how many weeks will be allotted to each one. Note that this schedule is tentative and is subjected to change as the semester progresses.

- **Mar 17, 2021 - Checkpoint 1 (1 week)** - Draw a schematic of your processor's datapath and pipeline stages, and provide a brief writeup of your answers to the questions in [2.21](#). In addition, push all of your IO-circuit Verilog modules that you have implemented in the labs to your assigned Github repository under `hardware/src/io_circuits` (see [2.2](#)). Also commit your design documents (block diagram + writeup) to `docs`.
- **April 14, 2021 - Checkpoint 2 (4 weeks)** - Implement a fully functional RISC-V processor core in Verilog. Your processor core should be able to run the `mmult` demo successfully.
- **April 28, 2021 - Checkpoint 3 (2 weeks)** - TBD
- **May 05, 2021 - Final Checkoff + Demo** - Final processor optimization and checkoff
- **May 07, 2021 - Project Report** - Final report due

1.2 General Project Tips

Document your project as you go. You should comment your Verilog and keep your diagrams up to date. Aside from the final project report (you will need to turn in a report documenting your project), you can use your design documents to help the debugging process.

Finish the required features first. Attempt extra features after everything works well. **If your submitted project does not work by the final deadline, you will not get any credit for any extra credit features you have implemented.**

This project, as has been done in past semesters, will be divided into checkpoints. The following sections will specify the objectives for each checkpoint.

2 Checkpoints 1 & 2 - 3-stage Pipelined RISC-V CPU

The first checkpoint in this project is designed to guide the development of a three-stage pipelined RISC-V CPU that will be used as a base system in subsequent checkpoints.

2.1 Setting up your Code Repository

The project skeleton files are available on Github. The suggested way for initializing your repository with the skeleton files is as follows:

```
git clone git@github.com:EECS150/project_skeleton_sp21.git
cd project_skeleton_sp21
git remote add my-repo git@github.com:EECS150/sp21_teamXX.git
git push my-repo main
```

Then reclone your repo and add the skeleton repo as a remote:

```
cd ..
rm -rf project_skeleton_sp21
git clone git@github.com:EECS150/sp21_teamXX.git
cd sp21_teamXX
git remote add staff git@github.com:EECS150/project_skeleton_sp21.git
```

To pull project updates from the skeleton repo, run `git pull staff main`.

To get a team repo, fill the [Google form](#) with your team information (names, Github logins). Only one person in a team is required to fill the form.

You should check frequently for updates to the skeleton files. Whenever you resume your work on the project, it is highly suggested that you do `git pull` from the skeleton repo to get the latest update. Update announcements will be posted to Piazza.

2.2 Integrate Designs from Labs

You should copy some modules you designed from the labs. We suggest you keep these with the provided source files in `hardware/src/io_circuits` (overwriting any provided skeletons).

Copy these files from the labs:

```
debouncer.v
synchronizer.v
edge_detector.v
```

fifo.v
uart_transmitter.v

2.3 Project Skeleton Overview

- hardware

- src

- * `z1top.v`: Top level module. The RISC-V CPU is instantiated here.
 - * `riscv_core/Riscv151.v`: All of your CPU datapath and control should be contained in this file.
 - * `io_circuits`: Your IO circuits from previous lab exercises.
 - * `EECS151.v`: Our EECS151-SP21 library file of register and memory modules. **You are expected to use these modules for your sequential logic.**

- sim

- * `Riscv151_testbench.v`: Starting point for testing your CPU. The testbench checks if your CPU can execute all the RV32I instructions (including CSR ones) correctly, and can handle some simple hazards. You should make sure that your CPU implementation passes this testbench before moving on.
 - * `assembly_testbench.v`: The testbench works with the software in `software/assembly_tests`.
 - * `isa_testbench.v`: The testbench works with the RISC-V ISA test suite in `software/riscv-isa-tests`.

The testbench only runs one test at a time. To run multiple tests, use the script we provide (see 2.14). There is a total of 38 ISA tests in the test suite.

- `echo_testbench.v`: The testbench works with the software in `software/echo`. The CPU reads a character sent from the serial rx line and echoes it back to the serial tx line.
 - `bios_testbench.v`: The testbench works with the BIOS program. The testbench checks if your CPU can execute the instructions stored in the BIOS memory. The testbench also emulates user input sent over the serial rx line, and checks the BIOS message output obtained from the serial tx line.
 - `strcmp_testbench.v`: The testbench works with the software in `software/strcmp`. This is an extra test for debugging.
 - `c_testbench.v`: The testbench works with the software in `software/c_test`. This is an extra test for debugging.

- software

- `bios151v3`: The BIOS program, which allows us to interact with our CPU via the UART. You need to compile it before creating a bitstream or running a simulation.

- **echo**: The echo program, which emulates the echo test of Lab 5 in software.
- **assembly_tests**: Use this as a template to write assembly tests for your processor designed to run in simulation.
- **c_example**: Use this as an example to write C programs.
- **riscv-isa-tests**: A comprehensive test suite for your CPU. Available after doing `git submodule` (see 2.14).
- **mmult**: This is a program to be run on the FPGA for Checkpoint 2. It generates 2 matrices and multiplies them. Then it returns a checksum to verify the correct result.

To compile **software** go into a program directory and run **make**. To build a bitstream run **make write-bitstream** in **hardware**.

2.4 RISC-V 151 ISA

Table 1 contains all of the instructions your processor is responsible for supporting. It contains most of the instructions specified in the RV32I Base Instruction set, and allows us to maintain a relatively simple design while still being able to have a C compiler and write interesting programs to run on the processor. For the specific details of each instruction, refer to sections 2.2 through 2.6 in the [RISC-V Instruction Set Manual](#).

2.4.1 CSR Instructions

You will have to implement 2 CSR instructions to support running the standard RISC-V ISA test suite. A CSR (or control status register) is some state that is stored independent of the register file and the memory. While there are 2^{12} possible CSR addresses, you will only use one of them (**tohost** = 0x51E). The **tohost** register is monitored by the RISC-V ISA testbench (**isa_testbench.v**), and simulation ends when a non-zero value is written to this register. A CSR value of 1 indicates success, and a value greater than 1 indicates which test failed.

There are 2 CSR related instructions that you will need to implement:

1. **csrw tohost,x2** (short for **csrrw x0,csr,rs1** where **csr** = 0x51E)
2. **csrwi tohost,1** (short for **csrrwi x0,csr,uimm** where **csr** = 0x51E)

csrw will write the value from **rs1** into the addressed CSR. **csrwi** will write the immediate (stored in the **rs1** field in the instruction) into the addressed CSR. Note that you do not need to write to **rd** (writing to **x0** does nothing), since the CSR instructions are only used in simulation.

2.5 Pipelining

Your CPU must implement this instruction set using a 3-stage pipeline. The division of the datapath into three stages is left unspecified as it is an important design decision with significant performance implications. We recommend that you begin the design process by considering which elements of the datapath are synchronous and in what order they need to be placed. After determining the design blocks that require a clock edge, consider where to place asynchronous blocks to minimize

Table 1: RISC-V ISA

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1	funct3		rd		opcode			R-type
imm[11:0]						rs1	funct3		rd		opcode			I-type
imm[11:5]				rs2		rs1	funct3		imm[4:0]		opcode			S-type
imm[12 10:5]				rs2		rs1	funct3		imm[4:1 11]		opcode			B-type
imm[31:12]									rd		opcode			U-type
imm[20 10:1 11 19:12]									rd		opcode			J-type

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

RV32/RV64 Zicsr Standard Extension

csr	rs1	001	rd	1110011	CSRRW
csr	uimm	101	rd	1110011	CSRRWI

the critical path. The RAMs we are using for the data, instruction, and BIOS memories are both **synchronous** read and **synchronous** write.

2.6 Hazards

As you have learned in lecture, pipelines create hazards. Your design will have to resolve both control and data hazards. You must resolve data hazards by implementing forwarding whenever possible. This means that you must forward data from your data memory instead of stalling your pipeline or injecting NOPs. All data hazards can be resolved by forwarding in a three-stage pipeline.

You'll have to deal with the following types of hazards:

1. **Read-after-write data hazards** Consider carefully how to handle instructions that depend on a preceding load instruction, as well as those that depend on a previous arithmetic instruction.
2. **Control hazards** What do you do when you encounter a branch instruction, a jal (jump and link), or jalr (jump from register and link)? You will have to choose whether to predict branches as taken or not taken by default and kill instructions that weren't supposed to execute if needed. You can begin by resolving branches by stalling the pipeline, and when your processor is functional, move to naive branch prediction.

2.7 Register File

We have provided a register file module for you in `EECS151.v: ASYNC_RAM_1W2R`. The register file has two asynchronous-read ports and one synchronous-write port (positive edge). In addition, you should ensure that register 0 is not writable in your own logic, i.e. reading from register 0 always returns 0.

2.8 RAMs

In this project, we will be using some memory blocks defined in `EECS151.v` to implement memories for the processor. As you may recall in previous lab exercises, the memory blocks can be either synthesized to Block RAMs or LUTRAMs on FPGA. For the project, our memory blocks will be mapped to Block RAMs. Therefore, read and write to memory are **synchronous**.

2.8.1 Initialization

For synthesis, the BIOS memory is initialized with the contents of the BIOS program, and the other memories are zeroed out.

For simulation, the provided testbenches initialize the BIOS memory with a program specified by the testbench (see `sim/assembly_testbench.v`).

2.8.2 Endianness + Addressing

The instruction and data RAMs have 16384 32-bit rows, as such, they accept 14 bit addresses. The RAMs are **word-addressed**; this means that every unique 14 bit address refers to one 32-bit row (word) of memory.

However, the memory addressing scheme of RISC-V is **byte-addressed**. This means that every unique 32 bit address the processor computes (in the ALU) points to one 8-bit byte of memory.

We consider the bottom 16 bits of the computed address (from the ALU) when accessing the RAMs. The top 14 bits are the word address (for indexing into one row of the block RAM), and the bottom two are the byte offset (for indexing to a particular byte in a 32 bit row).

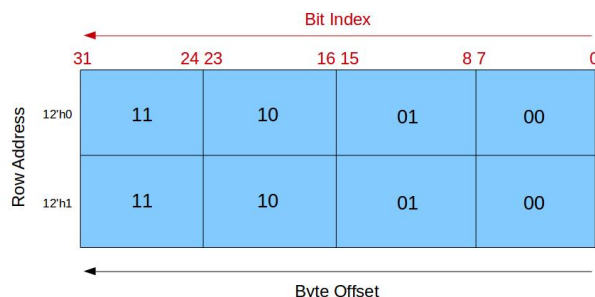


Figure 1: Block RAM organization. The labels for row address **should read 14'h0 and 14'h1**.

Figure 1 illustrates the 14-bit word addresses and the two bit byte offsets. Observe that the RAM organization is **little-endian**, i.e. the most significant byte is at the most significant memory address (offset '11').

2.8.3 Reading from RAMs

Since the RAMs have 32-bit rows, you can only read data out of the RAM 32-bits at a time. This is an issue when executing an `lh` or `lb` instruction, as there is no way to indicate which 8 or 16 of the 32 bits you want to read out.

Therefore, you will have to shift and mask the output of the RAM to select the appropriate portion of the 32-bits you read out. For example, if you want to execute a `lb` on a byte address ending in `2'b10`, you will only want bits `[23:16]` of the 32 bits that you read out of the RAM (thus storing `{24'b0, output[23:16]}` to a register).

2.8.4 Writing to RAMs

To take care of `sb` and `sh`, note that the `we` input to the instruction and data memories is 4 bits wide. These 4 bits are a byte mask telling the RAM which of the 4 bytes to actually write to. If `we={4'b1111}`, then all 32 bits passed into the RAM would be written to the address given.

Here's an example of storing a single byte:

- Write the byte `0xa4` to address `0x10000002` (byte offset = 2)
- Set `we = {4'b0100}`
- Set `din = {32'hxx_a4_xx_xx}` (x means don't care)

2.9 Memory Architecture

The standard RISC pipeline is usually depicted with separate instruction and data memories. Although this is an intuitive representation, it does not let us modify the instruction memory to run new programs. Your CPU, by the end of this checkpoint, will be able to receive compiled RISC-V binaries through the UART, store them into instruction memory, then jump to the downloaded program. To facilitate this, we will adopt a modified memory architecture shown in Figure 2.

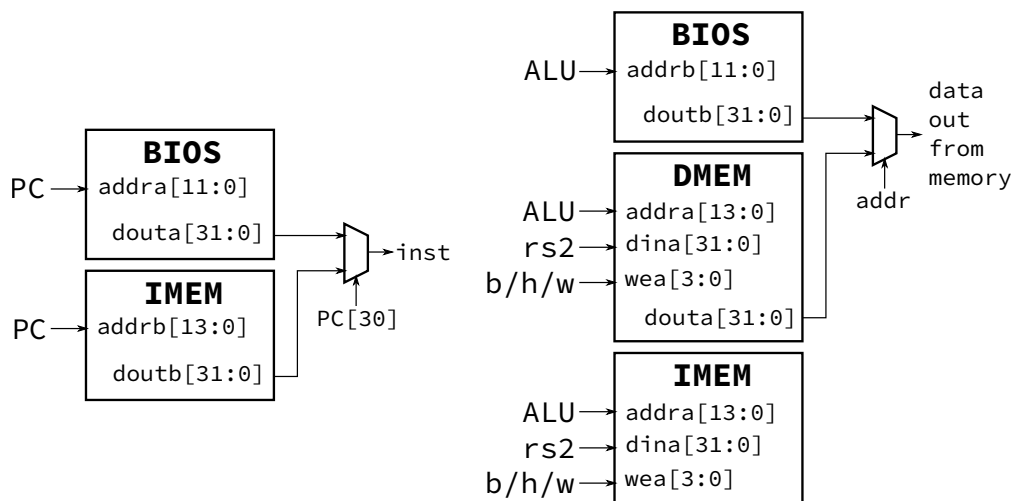


Figure 2: The Riscv151 memory architecture. There is only 1 IMEM and DMEM instance in Riscv151 but their ports are shown separately in this figure for clarity. The left half of the figure shows the instruction fetch logic and the right half shows the memory load/store logic.

2.9.1 Summary of Memory Access Patterns

The memory architecture will consist of three RAMs (instruction, data, and BIOS). The RAMs are memory resources (block RAMs) contained within the FPGA chip, and no external (off-chip, DRAM) memory will be used for this project.

The processor will begin execution from the BIOS memory, which will be initialized with the BIOS program (in `software/bios151v3`). The BIOS program should be able to read from the BIOS memory (to fetch static data and instructions), and read and write the instruction and data memories. This allows the BIOS program to receive user programs over the UART from the host PC and load them into instruction memory.

You can then instruct the BIOS program to jump to an instruction memory address, which begins execution of the program that you loaded. At any time, you can press the reset button on the board to return your processor to the BIOS program.

2.9.2 Unaligned Memory Accesses

In the official RISC-V specification, unaligned loads and stores are supported. However, in your project, you can ignore instructions that request an unaligned access. Assume that the compiler will never generate unaligned accesses.

2.9.3 Address Space Partitioning

Your CPU will need to be able to access multiple sources for data as well as control the destination of store instructions. In order to do this, we will partition the 32-bit address space into four regions: data memory read and writes, instruction memory writes, BIOS memory reads, and memory-mapped I/O. This will be encoded in the top nibble (4 bits) of the memory address generated in load and store operations, as shown in Table 2. In other words, the target memory/device of a load or store instruction is dependent on the address. The reset signal should reset the PC to the value defined by the parameter `RESET_PC` which is by default the base of BIOS memory (`0x40000000`).

Table 2: Memory Address Partitions

Address[31:28]	Address Type	Device	Access	Notes
4'b00x1	Data	Data Memory	Read/Write	
4'b0001	PC	Instruction Memory	Read-only	
4'b001x	Data	Instruction Memory	Write-Only	Only if PC[30] == 1'b1
4'b0100	PC	BIOS Memory	Read-only	
4'b0100	Data	BIOS Memory	Read-only	
4'b1000	Data	I/O	Read/Write	

Each partition specified in Table 2 should be enabled based on its associated bit in the address encoding. This allows operations to be applied to multiple devices simultaneously, which will be used to maintain memory consistency between the data and instruction memory.

For example, a store to an address beginning with `0x3` will write to both the instruction memory and data memory, while storing to addresses beginning with `0x2` or `0x1` will write to only the instruction or data memory, respectively. For details about the BIOS and how to run programs on your CPU, see Section 2.16.

Please note that a given address could refer to a different memory depending on which address type it is. For example the address `0x10000000` refers to the data memory when it is a data address while a program counter value of `0x10000000` refers to the instruction memory.

The note in the table above (referencing PC[30]), specifies that you can only write to instruction memory if you are currently executing in BIOS memory. This prevents programs from being self-modifying, which would drastically complicate your processor.

2.9.4 Memory Mapped I/O

At this stage in the project the only way to interact with your CPU is through the UART. The UART from Lab 5 accomplishes the low-level task of sending and receiving bits from the serial lines, but you will need a way for your CPU to send and receive bytes to and from the UART. To accomplish this, we will use memory-mapped I/O, a technique in which registers of I/O devices are assigned memory addresses. This enables load and store instructions to access the I/O devices as if they were memory.

To determine CPI (cycles per instruction) for a given program, the I/O memory map is also used to include instruction and cycle counters.

Table 3 shows the memory map for this stage of the project.

Table 3: I/O Memory Map

Address	Function	Access	Data Encoding
32'h80000000	UART control	Read	{30'b0, uart_rx_data_out_valid, uart_tx_data_in_ready}
32'h80000004	UART receiver data	Read	{24'b0, uart_rx_data_out}
32'h80000008	UART transmitter data	Write	{24'b0, uart_tx_data_in}
32'h80000010	Cycle counter	Read	Clock cycles elapsed
32'h80000014	Instruction counter	Read	Number of instructions executed
32'h80000018	Reset counters to 0	Write	N/A

You will need to determine how to translate the memory map into the proper ready-valid handshake signals for the UART. Your UART should respond to **sw**, **sh**, and **sb** for the transmitter data address, and should also respond to **lw**, **lh**, **lb**, **lhu**, and **lbu** for the receiver data and control addresses.

You should treat I/O such as the UART just as you would treat the data memory. This means that you should assert the equivalent write enable (i.e. valid) and data signals at the end of the execute stage, and read in data during the memory stage. The CPU itself should not check the `uart_rx_data_out_valid` and `uart_tx_data_in_ready` signals; this check is handled in software. The CPU needs to drive `uart_rx_data_out_ready` and `uart_tx_data_in_valid` correctly.

The cycle counter should be incremented every cycle, and the instruction counter should be incremented for every instruction that is committed (you should not count bubbles injected into the pipeline or instructions run during a branch mispredict). From these counts, the CPI of the processor can be determined for a given benchmark program.

2.10 Testing

The design specified for this project is a complex system and debugging can be very difficult without tests that increase visibility of certain areas of the design. In assigning partial credit at the end for incomplete projects, we will look at testing as an indicator of progress. A reasonable order in which to complete your testing is as follows:

1. Test that your modules work in isolation via Verilog testbenches
2. Test that your Riscv151 work with the `Riscv151_testbench.v`
3. Test the entire CPU one instruction at a time with hand-written assembly — see `assembly_testbench.v`
4. Run the `riscv-tests` ISA test suite
5. Some extra tests with other software C program, such as `c_test` and `strcmp`. They could help reveal more bugs — see `c_testbench.v` and `strcmp_testbench.v`
6. Test the CPU's memory mapped I/O — see `echo_testbench.v`
7. Test the CPU's memory mapped I/O with BIOS software program — see `bios_testbench.v`

2.11 Riscv151 Tests

Once you are confident that the individual components of your processor are working in isolation, you will want to test the entire processor as a whole. One way to do this is to pass the `Riscv151_testbench`. To run the test, use either one of the following commands (iverilog is highly recommended since it is faster):

```
# Simulate with sim/Riscv151_testbench.v
```

```
# with iverilog
```

```
make iverilog-sim tb=Riscv151_testbench
```

```
# open waveform
```

```
make wave tb=Riscv151_testbench
```

```
# with Vivado
```

```
make sim tb=Riscv151_testbench
```

The testbench covers all RV32I instructions. To pass this testbench, you should have a working Riscv151 implementation that can decode and execute all the instructions in the spec, including CSR instructions. Several hazard cases are also tested. The testbench does not work with any software code as in the following sections, but rather it manually initializes the instructions and data in the memory blocks as well as register file content for each test. The testbench does not cover reading from BIOS memory nor memory mapped IO. You will need to complete these components before moving on with other testbenches.

2.12 Software Toolchain - Writing RISC-V Programs

A GCC RISC-V toolchain has been built and installed in the `eecs151` home directory; these binaries will run on any of the `c125m` machines in the 125 Cory lab. The [VM Image](#) also has the toolchain installed along with Vivado 2019.1.

The most relevant programs in the toolchain are:

- `riscv64-unknown-elf-gcc`: GCC for RISC-V, compiles C code to RISC-V binaries.
- `riscv64-unknown-elf-as`: RISC-V assembler, compiles assembly code to RISC-V binaries.
- `riscv64-unknown-elf-objdump`: Dumps RISC-V binaries as readable assembly code.

Look at the `software/c_example` folder for an example of a C program.

There are several files:

- `start.s`: This is an assembly file that contains the start of the program. It initialises the stack pointer then jumps to the `main` label. Edit this file to move the top of the stack. Typically your stack pointer is set to the top of the data memory address space, so that the stack has enough room to grow downwards.

- `c_example.ld`: This linker script sets the base address of the program. For Checkpoint 2, this address should be in the format `0x1000xxxx`. The `.text` segment offset is typically set to the base of the instruction memory address space.
- `c_example.elf`: Binary produced after running `make`.
Use `riscv64-unknown-elf-objdump -Mnumeric -D c_example.elf` to view the assembly code.
- `c_example.dump`: Assembly dump of the binary.

2.13 Assembly Tests

Hand written assembly tests are in `software/assembly_tests/start.s` and the corresponding testbench is in `hardware/sim/assembly_testbench.v`. To run the test, run:

```
make sim tb=assembly_testbench
```

`start.s` contains assembly that's compiled and loaded into the BIOS RAM by the testbench.

```
_start:

# Test ADD
li x10, 100      # Load argument 1 (rs1)
li x11, 200      # Load argument 2 (rs2)
add x1, x10, x11 # Execute the instruction being tested
li x20, 1        # Set the flag register to stop execution and inspect the
↳ result register
                  # Now we check that x1 contains 300 in the testbench
```

Done: j Done

The `assembly_testbench` toggles the clock one cycle at time and waits for register `x20` to be written with a particular value (in the above example: 1). Once `x20` contains 1, the testbench inspects the value in `x1` and checks it is 300, which indicates your processor correctly executed the `add` instruction.

If the testbench times out it means `x20` never became 1, so the processor got stuck somewhere or `x20` was written with another value.

You should add your own tests to verify that your processor can execute different instructions correctly. Modify the file `start.s` to add your assembly code, then rerun the RTL simulation.

2.14 RISC-V ISA Tests

You will need the CSR instructions to work before you can use this test suite, and you should have confidence in your hand-written assembly tests. Test the CSR instructions using hand assembly tests.

To run the ISA tests, first pull the latest skeleton changes:

```
git pull staff main
git submodule update --init --recursive
```

Then run

```
cd hardware
```

```
# with iverilog
```

```
make iverilog-sim tb=isa_testbench test=all
```

```
# with Vivado
```

```
make sim tb=isa_testbench test=all
```

To run a particular ISA test (e.g. `add`), replace "all" with "add". The simulation should print out which tests passed or failed and their simulation cycles.

If you're failing other tests, debug using the test assembly file in `software/riscv-isa-tests/riscv-tests/isa/rv32ui` or the generated assembly dump. The assembly dump files are extremely helpful in debugging at this stage. If you look into a particular dump file of a test (e.g., `add.dump`), it contains several subtests in series. The CSR output from the simulation indicates which subtest is failing to help you narrow down where the problem is, and you can start debugging from there.

The `RESET_PC` parameter is used in `isa_testbench` to start the test in the IMEM instead of the BIOS. Make sure you have used it in `Riscv151.v`.

2.15 Software Tests: Echo

You should have your UART modules integrated with the CPU before running this test. The test verifies if your CPU is able to: check the UART status, read a character from UART Receiver, and write a character to UART Transmitter. Take a look at the software code `software/echo/echo.c` to see what it does. The testbench loads the MIF file compiled from the software code, and load it to the BIOS memory in a similar manner to the assembly test and riscv-isa tests.

To run the echo test, run

```
# with iverilog
```

```
make iverilog-sim tb=echo_testbench
```

```
# with Vivado
```

```
make sim tb=echo_testbench
```

The testbench, acts like a host, sends multiple characters via the serial line, then waits until it receives all the characters back. In some sense, it is similar to the echo test in Lab 5, however, the UART modules are controlled by the software program (`software/echo/echo.c`) running on your RISC-V CPU

In addition to the echo program, we also have other software programs such as

`software/c_test/c_test.c` and `software/strcmp_test/strcmp_test.c`.

which you can test with the following commands


```
# with iverilog
make iverilog-sim tb=c_testbench
make iverilog-sim tb=strcmp_testbench
```

```
# with Vivado
make sim tb=c_testbench
make sim tb=strcmp_testbench
```

These tests could help reveal more hazard bugs in your implementation. The tests use CSR instruction to indicate if they are passed (e.g., write '1' to CSR if passed). Take a look at the C files for further details. Following that practice, you can also write your custom C program to further test your CPU.

2.16 BIOS and Programming your CPU

We have provided a BIOS program in `software/bios151v3` that allows you to interact with your CPU and download other programs over UART. The BIOS is just an infinite loop that reads from the UART, checks if the input string matches a known control sequence, and then performs an associated action. For detailed information on the BIOS, see [Appendix B](#).

Before running the BIOS program on your FPGA, please do the final simulation test with the `sim/bios_testbench.v`. The testbench emulates the interaction between the host and your CPU via the serial lines orchestrated by the BIOS program.

```
# with iverilog
make iverilog-sim tb=bios_testbench
```

```
# with Vivado
make sim tb=bios_testbench
```

Once you pass the BIOS testbench, it's time to test it on the FPGA!

To run the BIOS:

1. Verify that the stack pointer and `.text` segment offset are set properly in `start.s` and `bios151v3.ld` in `software/bios151v3` directory
2. Build a bitstream and program the FPGA. Run `make write-bitstream` in `hardware` to generate a bitstream to your project, then `make program-fpga bs=bitstream_files/z1top.bit` to program the FPGA (if you are programming the FPGA from a lab machine with the Hardware Server, make sure that you update the port number in `hardware/scripts/program_fpga.tcl` to your assigned port number).

3. Use screen to access the serial port:

```
screen $SERIALTTY 115200
```

4. Press the reset button to make the CPU PC go to the start of BIOS memory

Close screen using `Ctrl-a Shift-k`, or other students won't be able to use the serial port! If you can't access the serial port you can run `killscreen` to kill all screen sessions.

If all goes well, you should see a `151 >` prompt after pressing return. The following commands are available:

- `jal <address>`: Jump to address (hex).
- `sw, sb, sh <data> <address>`: Store data (hex) to address (hex).
- `lw, lbu, lhu <address>`: Prints the data at the address (hex).

(if you want to backspace, press `Ctrl + Backspace`)

As an example, running `sw cafef00d 10000000` should write to the data memory and running `lw 10000000` should print the output `10000000: cafef00d`. Please also pay attention that writes to the instruction memory (`sw ffffffff 20000000`) do not write to the data memory, i.e. `lw 10000000` still should yield `cafef00d`.

In addition to the command interface, the BIOS allows you to load programs to the CPU. *With screen closed*, run:

```
hex_to_serial <mif_file> <address>
```

This stores the `.mif` file at the specified hex address. In order to write into both the data and instruction memories, **remember to set the top nibble to 0x3** (i.e. `hex_to_serial echo.mif 30000000`, assuming the `.ld` file sets the base address to `0x10000000`).

You also need to ensure that the stack and base address are set properly (See Section 2.12). For example, before making the `mmult` program you should set the set the base address to `0x10000000` (see 2.18). Therefore, when loading the `mmult` program you should load it at the base address: `hex_to_serial mmult.mif 30000000`. Then, you can jump to the loaded `mmult` program in in your screen session by using `jal 10000000`.

2.17 Target Clock Frequency

By default, the CPU clock frequency is set at 50MHz. It should be easy to meet timing at 50 MHz. Look at the timing report to see if timing is met. If you failed, the timing reports specify the critical path you should optimize.

For this checkpoint, we will allow you to demonstrate the CPU working at 50 MHz, but for the final checkoff at the end of the semester, you will need to optimize for a higher clock speed ($\geq 100\text{MHz}$) for full credit. Details on how to build your FPGA design with a different clock frequency will come later.

2.18 Matrix Multiply

To check the correctness and performance of your processor we have provided a benchmark in `software/mmult/` which performs matrix multiplication. You should be able to load it into your processor in the same way as loading the `echo` program.

This program computes $S = AB$, where A and B are 64×64 matrices. The program will print a checksum and the counters discussed in Section 2.9.4. The correct checksum is `0001f800`. If you do not get this, there is likely a problem in your CPU with one of the instructions that is used by the BIOS but not `mmult`.

The matrix multiply program requires that the stack pointer and the offset of the .text segment be set properly, otherwise the program will not execute properly.

The stack pointer (set in `start.s`) should start near the top of DMEM to avoid corrupting the program instructions and data. It should be set to `0x1000fff0` and the stack grows downwards.

The .text segment offset (set in `mmult.ld`) needs to accommodate the full set of instructions and static data (three 64×64 matrices) in the `mmult` binary. It should be set to the base of DMEM: `0x10000000`.

The program will also output the values of your instruction and cycle counters (in hex). These can be used to calculate the CPI for this program. Your target CPI should not be greater than 1.2. If your CPI exceeds this value, you will need to modify your datapath and pipeline to reduce the number of bubbles inserted for resolving control hazards (since they are the only source of extra latency in our processor). This might involve performing naive branch prediction or moving the `jalr` address calculation to an earlier stage.

2.19 How to Survive This Checkpoint

Start early and work on your design incrementally. Draw up a very detailed and organised block diagram and keep it up to date as you begin writing Verilog. Unit test independent modules such as the control unit, ALU, and regfile. Write thorough and complex assembly tests by hand, and don't solely rely on the RISC-V ISA test suite. The final BIOS program is several 1000 lines of assembly and will be nearly impossible to debug by just looking at the waveform.

The most valuable asset for this checkpoint will not be your GSIs but will be your fellow peers who you can compare notes with and discuss design aspects with in detail. However, do NOT under any circumstances share source code.

Once you're tired, go home and *sleep*. When you come back you will know how to solve your problem.

2.19.1 How To Get Started

It might seem overwhelming to implement all the functionality that your processor must support. The best way to implement your processor is in small increments, checking the correctness of your processor at each step along the way. Here is a guide that should help you plan out Checkpoint 1 and 2:

1. *Design*. You should start with a comprehensive and detailed design/schematic. Enumerate all the control signals that you will need. Be careful when designing the memory fetch stage since all the memories we use (BIOS, instruction, data, IO) are synchronous.
2. *First steps*. Implementing some modules that are easy to write and test.
3. *Control Unit + other small modules*. Implement the control unit, ALU, and any other small independent modules. Unit test them.
4. *Memory*. In the beginning, only use the BIOS memory in the instruction fetch stage and only use the data memory in the memory stage. This is enough to run assembly tests.

5. *Connect stages and pipeline.* Connect your modules together and pipeline them. At this point, you should be able to run integration tests using assembly tests for most R and I type instructions.
6. *Implement handling of control hazards.* Insert bubbles into your pipeline to resolve control hazards associated with JAL, JALR, and branch instructions. Don't worry about data hazard handling for now. Test that control instructions work properly with assembly tests.
7. *Implement data forwarding for data hazards.* Add forwarding muxes and forward the outputs of the ALU and memory stage. Remember that you might have to forward to ALU input A, ALU input B, and data to write to memory. Test forwarding aggressively; most of your bugs will come from incomplete or faulty forwarding logic. Test forwarding from memory and from the ALU, and with control instructions.
8. *Add BIOS memory reads.* Add the BIOS memory block RAM to the memory stage to be able to load data from the BIOS memory. Write assembly tests that contain some static data stored in the BIOS memory and verify that you can read that data.
9. *Add Inst memory writes and reads.* Add the instruction memory block RAM to the memory stage to be able to write data to it when executing inside the BIOS memory. Also add the instruction memory block RAM to the instruction fetch stage to be able to read instructions from the inst memory. Write tests that first write instructions to the instruction memory, and then jump (using jalr) to instruction memory to see that the right instructions are executed.
10. *Run Riscv151_testbench.* The testbench verifies if your Riscv151 is able to read the RV32I instructions from instruction memory block RAM, execute, and write data to either the Register File or data memory block RAM.
11. *Run isa_testbench.* The testbench works with the RISC-V ISA tests. This comprehensive test suites verifies the functionality of your processor.
12. *Add cycle counters.* Begin to add the memory mapped IO components, by first adding the cycle and instruction counters. These are just 2 32-bit registers that your CPU should update on every cycle and every instruction respectively. Write tests to verify that your counters can be reset with a `sw` instruction, and can be read from using a `lw` instruction.
13. *Integrate UART.* Add the UART to the memory stage, in parallel with the data, instruction, and BIOS memories. Detect when an instruction is accessing the UART and route the data to the UART accordingly. Make sure that you are setting the UART ready/valid control signals properly as you are feeding or retrieving data from it. We have provided you with the `echo_testbench` which performs a test of the UART. In addition, also run `bios_testbench`.
14. *Run the BIOS.* If everything so far has gone well, program the FPGA. Verify that the BIOS performs as expected. As a precursor to this step, you might try to build a bitstream with the BIOS memory initialized with the echo program.
15. *Run matrix multiply.* Load the `mmult` program with the `hex_to_serial` utility, and run `mmult` on the FPGA. Verify that it returns the correct checksum.
16. *Check CPI.* Compute the CPI when running the `mmult` program. If you achieve a CPI 1.2 or smaller, that is acceptable, but if your CPI is larger than that, you should think of ways to

reduce it.

2.20 Checkoff

The checkoff is divided into two stages: block diagram/design and implementation. The second part will require significantly more time and effort than the first one. As such, completing the block diagram in time for the design review is crucial to your success in this project.

2.20.1 Checkpoint 1: Block Diagram

The first checkpoint requires a detailed block diagram of your datapath. The diagram should have a greater level of detail than a high level RISC datapath diagram. You may complete this electronically or by hand.

If working by hand, we recommend working in pencil and combining several sheets of paper for a larger workspace. If doing it electronically, you can use Inkscape, Google Drawings, draw.io or any program you want.

You should be able to describe in detail any smaller sub-blocks in your diagram. **Though the diagrams from textbooks/lecture notes are a decent starting place, remember that they often use asynchronous-read RAMs for the instruction and data memories, and we will be using synchronous-read block RAMs.**

Additionally, you will be asked to provide short answers to the following questions based on how you structure your block diagram. The questions are intended to make you consider all possible cases that might happen when your processor execute instructions, such as data or control hazards. It might be a good idea to take a moment to think of the questions first, then draw your diagram to address them.

2.21 Questions

1. How many stages is the datapath you've drawn? (i.e. How many cycles does it take to execute 1 instruction?)
2. How do you handle ALU \rightarrow ALU hazards?
`addi x1, x2, 100`
`addi x2, x1, 100`
3. How do you handle ALU \rightarrow MEM hazards?
`addi x1, x2, 100`
`sw x1, 0(x3)`
4. How do you handle MEM \rightarrow ALU hazards?
`lw x1, 0(x3)`
`addi x1, x1, 100`

5. How do you handle MEM \rightarrow MEM hazards?

```
lw x1, 0(x2)
```

```
sw x1, 4(x2)
```

also consider:

```
lw x1, 0(x2)
```

```
sw x3, 0(x1)
```

6. Do you need special handling for 2 cycle apart hazards?

```
addi x1, x2, 100
```

```
nop
```

```
addi x1, x1, 100
```

7. How do you handle branch control hazards? (What is the mispredict latency, what prediction scheme are you using, are you just injecting NOPs until the branch is resolved, what about data hazards in the branch?)
8. How do you handle jump control hazards? Consider jal and jalr separately. What optimizations can be made to special-case handle jal?
9. Can you handle jal writeback data forwarding?

```
jal x1, label
```

```
label: addi x2, x1, 4
```

10. How do you resolve JALR? How many NOPs are injected?
11. What is the most likely critical path in your design?
12. Where do the UART and cycle counters go? How are you going to drive data_in_valid (give a logic expression)?
13. Where does the BIOS mem go? How about the IMEM (in the memory stage)?

Commit your block diagram and your writeup to your team repository under `sp21_teamXX/docs` by Mar 17, 2021. Please also remember to push your working IO circuits to your Github repository.

2.21.1 Checkpoint 2: Base RISC-V151 System

This checkpoint requires a fully functioning three stage RISC-V CPU as described in this specification. Checkoff will consist of a demonstration of the BIOS functionality, loading a program (`echo` and `mmult`) over the UART, and successfully jumping to and executing the program.

Additionally, please find the maximum achievable frequency of your CPU implementation. To do so, lower the `CPU_CLOCK_PERIOD` (starting at 20, with a step size of 1) in `hardware/src/z1top.v` until the Implementation fails to meet timing. Please report the critical path in your implementation.

Checkpoint 2 materials should be committed to your project repository by April 14, 2021.

2.21.2 Checkpoints 1 & 2 Deliverables Summary

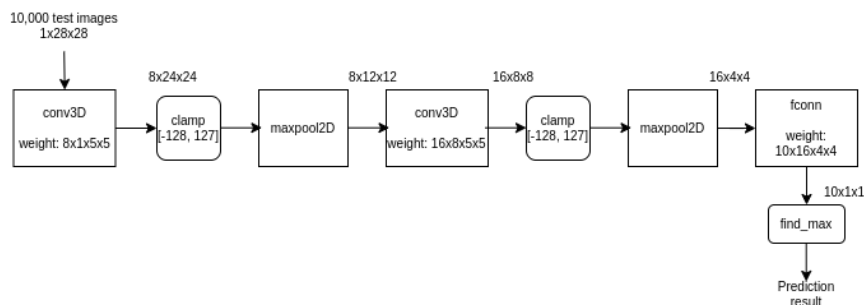
Deliverable	Due Date	Description
Block Diagram, RISC-V ISA Questions, IO code	Mar 17, 2021	Push your block diagram, your writeup, and IO code to your Github repository. In-lab Checkoff: Sit down with a GSI and go over your design in detail.
RISC-V CPU, Fmax and Crit. path	April 14, 2021	Check in code to Github. In-lab Checkoff: Demonstrate that the BIOS works, you can use <code>hex_to_serial</code> to load the <code>echo</code> program, <code>jal</code> to it from the BIOS, and have that program successfully execute. Load the <code>mmult</code> program with <code>hex_to_serial</code> , <code>jal</code> to it, and have it execute successfully and return the benchmarking results and correct checksum. Your CPI should not be greater than 1.2

3 Checkpoint 3 - Hardware-Accelerated Convolutional Neural Network

3.1 Checkpoint Overview

3.1.1 LeNet

In this checkpoint, your task is to design and implement an optimized hardware accelerator to speed up the inference execution of a small Convolutional Neural Network (CNN). You will be comparing the performance of your accelerator against the CNN software implementation running on the RISC-V processor that you have implemented in Checkpoint 2. Our target CNN is the classic LeNet-5 network used for handwritten digit classification. LeNet-5 is a feed-forward network that consists of 5 layers as shown in the following figure.



A rigorous understanding of deep neural networks is not required when you do this checkpoint, since we will provide the software model of the problem. Nonetheless, if you'd like to look at some introductory materials to get yourself acquainted with some Deep learning concepts, the [Deep Neural Networks Design and Examples](#) lecture or the lecture notes from [EE290-2](#) are good places to start.

Here are some key parameters of the network. We refer an input matrix to a layer as input feature map (IFM), and the output matrix of a layer as output feature map (OFM).

- The first 3D Convolutional layer (conv3D): input feature map $1 \times 28 \times 28$ (8-bit), weight $8 \times 1 \times 5 \times 5$ (8-bit), output feature map $8 \times 24 \times 24$ (32-bit)
- The first 2D Max Pooling layer (maxpool2D): input feature map $8 \times 2 \times 24$ (32-bit), output feature map $8 \times 12 \times 12$ (8-bit)
- The second 3D Convolutional layer: input feature map $8 \times 12 \times 12$ (8-bit), weight $16 \times 8 \times 5 \times 5$ (8-bit), output feature map $16 \times 8 \times 8$ (32-bit)
- The second 2D Max Pooling layer: input feature map $16 \times 8 \times 8$ (32-bit), output feature map $16 \times 4 \times 4$ (8-bit)
- The Fully-Connected layer (fconn): input feature map $16 \times 4 \times 4$ (8-bit), weight $10 \times 16 \times 4 \times 4$ (8-bit), output feature map $10 \times 1 \times 1$ (32-bit)

The network is trained with the MNIST dataset to obtain the weight data for the two conv3D layers and fconn layers. The weights, initially in floating-point datatype, are then quantized to 8-bit signed integers (-128 to 127) to reduce the implementation complexity (no floating-point

computation required) as well as make the network leaner and more FPGA-friendly. The general consensus is that the inference does not need a lot of bits to achieve competitive accuracy to full precision as opposed to the training process. The output feature maps of these layers are also quantized to 8-bit signed integers by being scaled down by a factor of 512 and clamped to within the range of -128 to 127. The quantization has negligible impact on the accuracy of the network (97% on 10000 MNIST test images). This technique is called Post-training quantization. The Python script for training the network is adapted from the [Lab 1 material](#) of the course EE290-2.

You might have noticed that the network is small enough that we could fit all the weight data of these layers on our PYNQ-Z1. This is not always possible for state-of-the-art neural networks. To make the problem more challenging and practical, the quantized weights, along with the 10000 quantized test images are initialized on the off-chip DDR of the PYNQ-Z1 platform. Therefore, data won't simply be initialized in any on-chip Memory blocks and embedded in the bitstream as we've done so far, instead extra efforts must be done to bring the data closer to your computational units.

This checkpoint gives you a different design problem from what was asked in checkpoint 2, since now you will need to build a specialized datapath for computation and memory buffering optimized specifically for this application. You are free to use any modules you like (memory blocks, FIFOs) for your design as long as everything is working correctly and faster than the baseline (naive) implementation. There is also no restriction on the amount of memory storage for your accelerator design.

From the previous lab and homework exercises, you have already had some practice on mapping an algorithmic description of a problem (e.g., loop forms) into a circuit implementation. This checkpoint brings it to a grander level by adding more complexity in terms of the number of loop levels and the amount of operations needed to carry out per loop iteration. How do we make sense between a software loop iteration and a hardware cycle? Would it be possible to map one or more software loop iterations to one hardware cycle? Can we overlap the operations of current loop iteration with the next one? Can we unroll the loops to achieve higher degree of parallelism? Can we bank a memory module such that we are not constrained by the number of available memory ports for parallel read/write? Bring all the design techniques that you have learned from lectures, and see how far you can get with your accelerator design.

3.1.2 New files

Please do **git pull staff main** to get the latest updates for this checkpoint from the staff repository. Some new files have been added:

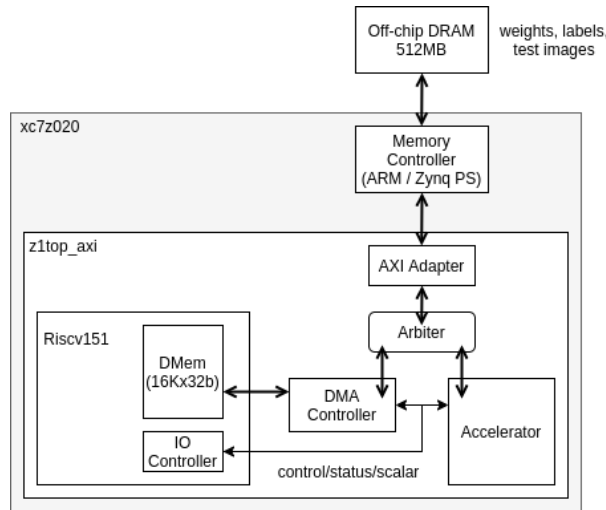
- `hardware/src/z1top_axi.v`: New top-level module to integrate with the Zynq Processing System IP. We will use this top module from now on for the project.
- `hardware/src/accelerator/xcel_naive.v`: An accelerator with a naive conv3D implementation. You will use this as the baseline.
- `hardware/src/accelerator/xcel_opt.v`: A skeleton code for the optimized accelerator module. **Your task is to complete this file.**
- `hardware/src/accelerator/axi_mm_adapter.v`: An AXI Memory-Mapped (MM) master

adapter core for interfacing with the Zynq Processing System.

- `hardware/src/accelerator/axi_mm_read.v`: A sub-component of the AXI MM adapter that implements the AXI read logic.
- `hardware/src/accelerator/axi_mm_write.v`: A sub-component of the AXI MM adapter that implements the AXI write logic.
- `hardware/src/accelerator/dma_controller.v`: A DMA (Direct-Memory Access) controller for interfacing with the off-chip DDR and the RISC-V Data Memory (DMem) via the AXI adapter.
- `hardware/src/accelerator/arbiter.v`: An arbiter for selecting which client (DMA or Accelerator) to service requests and responses to and from the off-chip DDR.
- `hardware/sim/xcel_testbench.v`: A testbench for verifying the functionality of the `xcel` implementation. Only conv3D operation is tested.
- `hardware/sim/mem_model.v`: A simple memory model that works with the `xcel_testbench.v`.
- `software/axi_test/*`: Software files for testing AXI communication (read and write).
- `software/lenet/*`: Software files for the LeNet inference demo.

3.1.3 High-level overview of the full system

The following figure shows the big picture of the full system.



The DMA Controller orchestrates the memory communication between the RISC-V's DMem and the off-chip DDR. The AXI adapter receives a write or read request from either the DMA or the Accelerator, and submits the request to the off-chip DDR through the Zynq Processing System. The arbiter implements a simple MUXing logic to service the request and response from the Accelerator if it is currently running, otherwise the DMA. Note that there is no direct communication between the RISC-V and the Accelerator. All communication must go through the off-chip DDR. For example, if the Accelerator wants to read some data from the RISC-V's DMem block, it must

be first transferred to a memory location of the off-chip DDR by the DMA, and the data can be accessed by the read logic at the specified DDR address implemented inside the Accelerator. Similarly, any result data computed by the Accelerator must also be written to the DDR before the RISC-V core can read it. The operations of the DMA and the Accelerator are controlled by the IO controller inside the Riscv151 core. To the Riscv151 core's perspective, they act as IO devices as similar to the UART modules.

3.1.4 ARM Baremetal Application

A pre-compiled ARM binary is provided to initialize the off-chip DRAM with the weight, image, and label data. Please run the following command to clone the submodule to your repository.

You can take a look at the C source file `arm_baremetal_app/system/helloworld.c` to see how things are set up. In order to compile this application, you will need to have access to a different Xilinx software (Vivado SDK). However, since the application is pre-compiled, you do not need to worry about this. You don't even need to touch any of the files in this submodule when doing the checkpoint. The addresses of the weight, image, and label data are statically known and fixed. We will provide these addresses to the DMA Controller and the Accelerator so that they can access to the correct locations of the DDR to get the data.

3.2 Software Implementation

This section provides the software description for the computational kernels found in our network: conv3D, maxpool2D, and fconn. Their implementations are slightly different from the conventional ones that you might have seen in other resources since our network is quantized to 8-bit signed integers.

You are not required to implement all these computational kernels in your accelerator, although you're welcome to do so. The bare minimum requirement is that you must accelerate the 3D Convolutional function as similar to the naive accelerator implementation given to you.

To keep things simple, your conv3D module is only required to handle convolution with a stride of 1 and no padding. Therefore, no boundary check is necessary. In addition, the weight dimension is passed as a parameter to the Verilog module (statically known). You're encourage to write your code in a parameterizable manner since it is a good coding practice (i.e., no assumption on the value of the weight dimension), but this is not strictly compulsory.

3.2.1 conv3D

```
// Since the stride is 1 and no padding required, OFM_DIM = IFM_DIM - WT_DIM + 1

#define OFM_SIZE  (OFM_DIM * OFM_DIM)
#define IFM_SIZE  (IFM_DIM * IFM_DIM)
#define WT_SIZE   (WT_DIM * WT_DIM)
#define WT_VOLUME (IFM_DEPTH * WT_SIZE)

for (f = 0; f < OFM_DEPTH; f++) {
```

```

for (i = 0; i < OFM_DIM; i++) {
    for (j = 0; j < OFM_DIM; j++) {
        ofm[f * OFM_SIZE + i * OFM_DIM + j] = 0;
    }
}

for (d = 0; d < IFM_DEPTH; d++) {
    for (i = 0; i < OFM_DIM; i++) {
        for (j = 0; j < OFM_DIM; j++) {
            int32_t tmp = 0;
            for (m = 0; m < WT_DIM; m++) {
                for (n = 0; n < WT_DIM; n++) {
                    int32_t ifm_data = ifm[d * IFM_SIZE + (i + m) * IFM_DIM + (j + n)];
                    int32_t wt_data = wt[f * WT_VOLUME + d * WT_SIZE + m * WT_DIM + n];
                    tmp += ifm_data * wt_data;
                }
            }

            ofm[f * OFM_SIZE + i * OFM_DIM + j] += tmp >> 9;
        }
    }
}

```

3.2.2 maxpool2D

```

#define OFM_DIM (IFM_SIZE / 2)
#define OFM_SIZE (OFM_DIM * OFM_DIM)
#define IFM_SIZE (IFM_DIM * IFM_DIM)

for (f = 0; f < OFM_DEPTH; f++) {
    for (i = 0; i < OFM_DIM; i++) {
        for (j = 0; j < OFM_DIM; j++) {
            int8_t tmp0 = ifm[d * IFM_SIZE + (i*2+0) * IFM_DIM + (j*2+0)];
            int8_t tmp1 = ifm[d * IFM_SIZE + (i*2+1) * IFM_DIM + (j*2+0)];
            int8_t tmp2 = ifm[d * IFM_SIZE + (i*2+0) * IFM_DIM + (j*2+1)];
            int8_t tmp3 = ifm[d * IFM_SIZE + (i*2+1) * IFM_DIM + (j*2+1)];

            tmp0 = (tmp0 > 0) ? tmp0 : 0;
            tmp1 = (tmp1 > 0) ? tmp1 : 0;
            tmp2 = (tmp2 > 0) ? tmp2 : 0;
            tmp3 = (tmp3 > 0) ? tmp3 : 0;

            ofm[f * OFM_SIZE + i * OFM_DIM + j] = max(tmp0, tmp1, tmp2, tmp3);
        }
    }
}

```

```
}
```

3.2.3 fconn

```
// In some sense, a fully-connected operation can also be view as conv3D
// with WT_DIM == IFM_DIM (and OFM_DIM = 1)

#define WT_DIM      (IFM_DIM)
#define IFM_SIZE    (IFM_DIM * IFM_DIM)
#define WT_SIZE     (WT_DIM * WT_DIM)
#define WT_VOLUME   (IFM_DEPTH * WT_SIZE)

for (f = 0; f < OFM_DEPTH; f++) {
    int32_t tmp = 0;
    for (d = 0; d < IFM_DEPTH; d++) {
        for (i = 0; i < IFM_DIM; i++) {
            for (j = 0; j < IFM_DIM; j++) {
                int32_t ifm_data = ifm[d * IFM_SIZE + i * IFM_DIM + j];
                int32_t wt_data  = wt[f * WT_VOLUME + d * WT_SIZE + i * WT_DIM + j];
                tmp += ifm_data * wt_data;
            }
        }
    }
    ofm[f] += tmp;
}
```

3.3 Naive conv3D: Sources of Inefficiency

A straight mapping from the software description to hardware implementation, without considering many inherent characteristics of the system (memory latency, buffering, available compute blocks, etc.), undoubtedly yields poor result. Have a look at the baseline conv3D implementation given to you: `hardware/src/accelerator/xcel_naive.v`. There are a few weaknesses in this implementation.

- This implementation reads the elements of input feature map redundantly without considering the fact that there are some overlap between the current and the next sliding window (with a stride of 1). Therefore, the off-chip memory latency quickly overwhelms the performance and leaves the compute unit starving for data most of the time
- This implementation does not consider the fact that the weight and input feature map are 8-bit data, and that 4 weight/ifm items could be packed in a single read transfer at a time since our AXI data bus is 4 times wider. Instead, it issues 4 separate single read transfers for those consecutive weight/ifm data items. The redundancy adds extra communication overhead.
- This implementation does not utilize burst mode to achieve better memory bandwidth utilization, especially since there are multiple data items in a row and one could set up a burst

request to read them all in successive cycles with some initial startup overhead from the request to the response.

- This implementation does not utilize any intermediate buffer to store the partial compute results on chip. Instead, the partial output channel result (ofm) is written back to the DDR, and then reread to accumulate with the next result.
- Per sliding window computation, the weight and the ifm data are preloaded into shift registers, and perform a single multiply-accumulation (MAC) per cycle of the sliding window. However, one could also fully unroll this computation to use more MAC operations to compute a sliding window in a cycle with the trade-off of more hardware resource.
- The FSM design serializes the execution and leaves small room to achieve pipelining or overlapping between memory fetching/writing and computation. The FSM embeds the mental model of the sequential software execution. Can we redesign or get rid of the FSM to achieve better pipelining execution of memory fetch/write and computation?

To build an efficient accelerator, one needs to not only understand how the software code works, but also be able to transform it to some form that is hardware-friendly in terms of memory access patterns, data reuses, and parallel executions. The topic of CNN acceleration has been extensively studied and explored, so we won't have any issues of finding some existing architecture to implement for the checkpoint. Nonetheless, you're also welcome to pursue your own ideas.

3.4 AXI Bus Interface

This section intends to provide you some background on the ARM's AMBA AXI bus interface. AXI is a widely popular standard interface for establishing and standardizing the communication between many IP cores. Our communication cores (DMA, Accelerator, AXI Adapter) adopt the AXI4 interface bus protocol. There are separate read/write request and response channels. Per each channel, a handshake mechanism (Ready/Valid) is used to indicate a successful transaction.

You can skim the official ARM's AXI specification [here](#) for more information (chapter 2, 3, and 4). The reading is entirely optional. The following subsections give a brief summary on the protocol and things that are particularly important for this checkpoint. Additional resource is Xilinx [AXI Reference Guide](#).

To reduce the design complexity, our AXI data bus width is set to 32-bit to match with the data word width of the RISC-V processor. And similar to our processor core, the Zynq PS uses byte-level addressing. A read from a DDR memory location retrieves 4 bytes on the data bus. Likewise, a write to a memory location will update all 4 bytes. There's also a write strobe signal to set which byte lane(s) of the data bus to write to the memory.

The AXI read/write logic to interface with the Zynq PS is provided to you:

`hardware/src/accelerator/axi_mm_adapter.v`.

One of your responsibilities in `hardware/src/accelerator/xcel_opt.v` is to figure out how to set up the read and write interface channels to the AXI Adapter core so that your accelerator can read or write data correctly from and to the off-chip DDR.

3.4.1 Read Interface Channels

1. Read Address Request channel

- **araddr**: Memory read request address (output)
- **arvalid**: Memory read request address valid signal (output)
- **arready**: Memory read request address ready signal (input)
- **arlen**: Memory read burst length (output): the number of data transfers per read transaction (offset by 1).
- **arsize**: Memory read burst size (output): the number of bytes per data transfer (log based 2).
- **arburst**: Memory read burst type (output). Set to *INCREMENT* so that the bus receives successive data items in a row.

2. Read Data Response channel

- **rdata**: Memory read response data (input)
- **rvalid**: Memory read response data valid signal (input)
- **rready**: Memory read response data ready signal (output)
- **rlast**: Memory read response data last signal (input). Stays HIGH on the last data transfer.

Figure 3 demonstrates an example of an AXI read transaction. A read request is submitted, and a burst of 4 data beats are sent to the read response data channel.

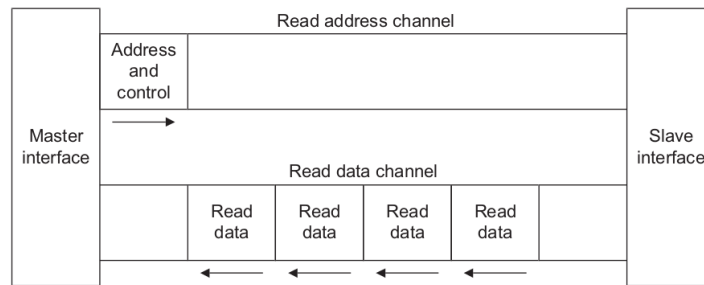


Figure 1-1 Channel architecture of reads

Figure 3: An AXI read transaction. Source: [AXI Spec](#)

Figure 4 shows the timings of the related AXI signals on a read transaction. Note how handshake is used to indicate when we have a fired request or response data (valid and ready are both HIGH at a rising clock edge).

3.4.2 Write Interface Channels

1. Write Address Request channel

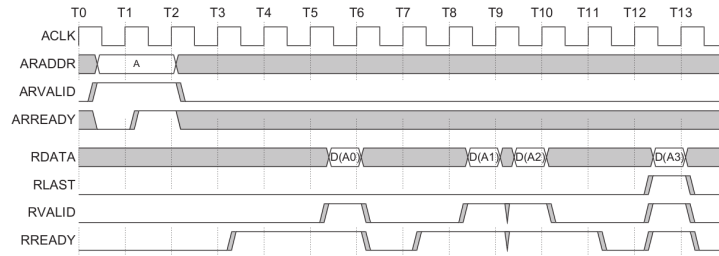


Figure 1-4 Read burst

Figure 4: An AXI read burst timing diagram. Source: [AXI Spec](#)

- **awaddr**: Memory write request address (output)
- **awvalid**: Memory write request address valid signal (output)
- **awready**: Memory write request address ready signal (output)
- **awlen**: Memory write burst length (output): the number of data transfers per read transaction (offset by 1).
- **awsize**: Memory write burst size (output): the number of bytes per data transfer (log based 2).
- **awburst**: Memory write burst type (output). Set to *INCREMENT* so that the bus receives successive data items in a row.

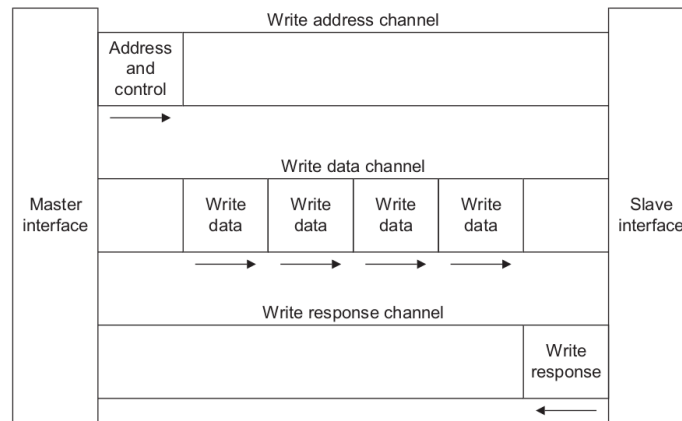


Figure 1-2 Channel architecture of writes

Figure 5: An AXI write transaction. Source: [AXI Spec](#)

2. Write Data Request channel

- **wdata**: Memory write request data (output)
- **wvalid**: Memory write request data valid (output)

- **wready**: Memory write request data ready (input)
- **wlast**: Memory write request data last (output). Set to HIGH on the last write data transfer.
- **wstrb**: Memory write request data strobe (output). Set to 4'b1111 if writing the full word to the DDR memory.

3. Write Response channel

- **bresp**: Memory write response (input). The value *RESP_OKAY* (4'b0000) indicates a write success.
- **bvalid**: Memory write response valid signal (input)
- **bready**: Memory write response ready signal (output)

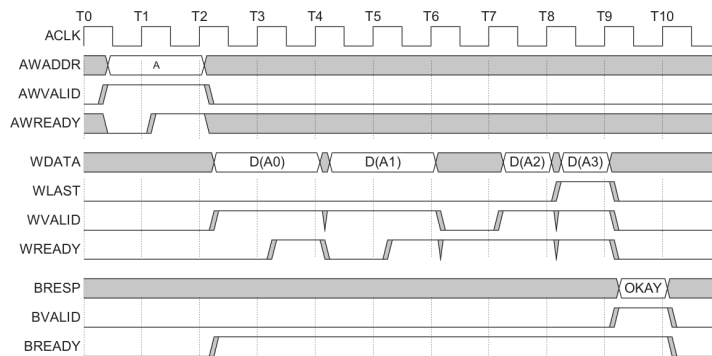


Figure 1-6 Write burst

Figure 6: An AXI write burst timing diagram. Source: [AXI Spec](#)

Figure 5 demonstrates an example of an AXI write transaction. A write request is submitted, and a burst of 4 data beats are sent to the write data channel.

Figure 6 shows the timings of the related AXI signals on a write transaction. Also note how handshake is used to indicate when we have a fired request or response.

3.5 Vivado Block Design with Zynq Processing System (PS)

Figure 7 shows the Block design of the full system after running the following command.

```
make build-project proj=z1top_axi
```

Our `z1top.v` communicates with the Zynq PS through the AXI SmartConnect IP using the AXI4 bus interface. The Zynq PS also serves as a memory controller to the off-chip DDR. Therefore, the only way for a module implemented in the Programmable Logic (PL) to access to the DDR is through the Zynq PS. A High Performance AXI port (HP0) is used to communicate with the PL. The Zynq PS can be configured to use up to 4 HPs (HP0-3) to increase the memory bandwidth. For the sake of simplicity, we only use one HP port.

Also note that the Zynq PS supplies the clock signal `FCLK_CLK0` to the fabric modules. To avoid extra timing complexity concerning with multi-clock domains, we will not use our usual

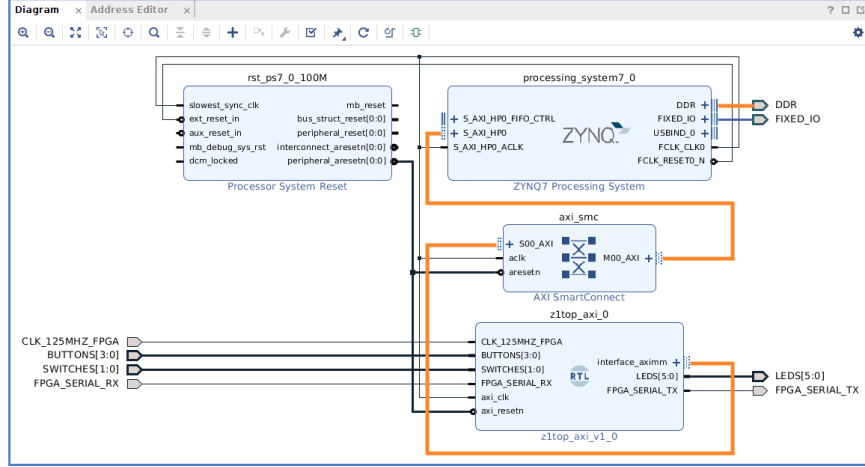


Figure 7: The Block Design of z1top-axi module with the Zynq PS

CLK_FPGA_125MHZ, but rather rely on the clock signal from the PS to clock the synchronous logic of our implementation.

3.6 System Integration with Riscv151

3.6.1 IO Memory-mapped Addresses for DMA Controller and Accelerator

You will need to extend your IO memory-mapped logic to integrate the DMA controller and the Accelerator (xcel) cores to Riscv151. We will use the load and store instructions to the memory-mapped addresses as a mechanism to control (start, reset) or check the status of these blocks (idle, done). The done signal should be kept HIGH once the dma or xcel finishes, since a software code will keep polling on this signal (busy-waiting) before moving to the next part of the program. In addition, we introduce additional memory-mapped registers to set various address offsets so that the cores know where to access the correct data in any memory devices (DDR or DMem). This is a neat approach to avoid hard-coding the addresses or recompiling the whole bitstream everytime we change the memory layout of the arrays. The input/output feature map dimension and depth can also be set in software code.

The memory-mapped addresses are defined in `software/151_library/memory_map.h`.

3.6.2 Integrating DMA Controller with CPU Data Memory

You might already notice that this system would not work with the current setting of Riscv151 in that DMem is a single-port memory block. In addition, port a is currently being used for the CPU load and store instructions. We would need to do the following steps to integrate the DMA Controller safely to the existing system.

- Use a dual-port memory template for DMem (look at IMem as an example)
- Connect port b of DMem to the Memory interface of the DMA Controller (via Riscv151).

This will enable the DMA Controller to have access to the Data Memory.

Table 4: I/O Memory Map for the DMA Controller and Accelerator

Address	Function	Access	Data Encoding
32'h80000030	dma control (start)	Write	N/A
32'h80000034	dma status	Read	{30'b0, idle, done}
32'h80000038	dma direction (1: DMem to DDR, 0: DDR to DMem)	Write	IO store data (32-bit)
32'h8000003c	dma source address	Write	IO store data (32-bit)
32'h80000040	dma destination address	Write	IO store data (32-bit)
32'h80000044	dma transfer length (per 4 bytes)	Write	IO store data (32-bit)
32'h80000050	xcel control (start)	Write	N/A
32'h80000054	xcel status	Read	{30'b0, idle, done}
32'h80000058	xcel input feature map DDR address	Write	IO store data (32-bit)
32'h8000005c	xcel weight DDR address	Write	IO store data (32-bit)
32'h80000060	xcel output feature map DDR address	Write	IO store data (32-bit)
32'h80000064	xcel input feature map dimension	Write	IO store data (32-bit)
32'h80000068	xcel input feature map depth (channel)	Write	IO store data (32-bit)
32'h8000006c	xcel output feature map dimension	Write	IO store data (32-bit)
32'h80000070	xcel output feature map depth (channel)	Write	IO store data (32-bit)

3.7 LeNet Demo software code

...

3.8 Resources and Suggestions for Accelerator design

You are encouraged to visit the [MIT DNN Eyeriss tutorial](#) for ideas on how to build the accelerator for the 3D Convolution operation or CNN in general. The DNN Accelerators slides: [Part 1](#) and [Part 2](#) are particularly informative and might be useful for you as a starting point (check for *Row-stationary architecture*). If you need a detailed cycle-by-cycle explanation, you can have a look at the slides here ...

3.9 High-throughput Spatially-mapped Design (Extra Credit)

...

3.10 Steps to Complete Checkpoint 3

First, get the latest changes from the project skeleton repo

```
git pull staff main
git submodule update --init --recursive
```

This will clone the `arm_baremetal_app` submodule.

The next step is integrating the `dma_controller` and `xcel_naive` to your existing Riscv151 processor. You need to expand your IO memory-mapped logic to perform load/store to the memory addresses of these blocks as mentioned in the section above, so that the CPU can control them from software. In addition, you will also need to convert `DMem` to dual-port memory block. Use `port b` to

connect to the `dma_controller` via the interface of the Riscv151 processor. Also note that we're no longer using `hardware/src/z1top.v` for this checkpoint, but `hardware/src/z1top_axi.v` instead.

Next, generate a bitstream to configure your FPGA. Now you can pass the target clock frequency to the command.

```
# This command will implement and generate bitstream  
# for the z1top_axi module and set the clock period to 20ns  
make write-bitstream proj=z1top_axi clk=20
```

Next, before programming the FPGA, you need to run the following script to initialize the Zynq PS (init the clock signal and the DDR).

```
# If you are programming the FPGA from a lab machine with the Hardware Server,  
# make sure that you update the port number in  
# hardware/scripts/init_arm.tcl to your assigned port number.  
# This script only needs to run once when you first turn on the board,  
# or when your bitstream is implemented with a new clock period  
make init-arm
```

```
# Program the board  
make program-fpga bs=bitstream_files/z1top_axi.bit
```

Next, test the DDR communication with your Riscv151 to verify that you have integrated the DMA correctly. There is a software program for that `software/axi_test`. Run the program using the same command as in `mmult` case. Alternatively, you can do

```
cd software/axi_test  
make run
```

This program sends an array allocated in `DMem` of your RISC-V core to a memory location in the DDR (DMA write operation), and read from that memory location (DMA read operation) and write the result to a different array in `DMem`. There should not be any mismatches between the two arrays once the DMA finishes. Try testing with different transfer lengths or addresses.

If the DMA is working, you can move on to the next part, which is to run the LeNet software. Go to `software/lenet`, and run the program as follows.

```
# This will compile and run the software implementation of LeNet  
make xcel=SW  
make run  
# This will compile and run the hardware implementation of LeNet (only conv3D is  
# put in hardware)  
make xcel=HW  
make run
```

The program tests 128 images with 127 correct predictions (`0000_007f`). The failed prediction is at the image `0000_0073` with a prediction of 9 while the groundtruth is 4. The groundtruth labels can be checked [here](#).

You can also test with fewer images (change the macro `NUM_TEST_IMAGES` in `lenet.c`) to make the program run a little faster.

Once you get a sense of how the entire flow works, it's time to code your own accelerator! You can use the naive code as a starting point, but please feel free to write your own code for everything. Add your own implementation to `xcel_opt.v`, then replace the naive module with the opt module in `z1top_axi.v`.

As an additional tip for debugging, you can take advantage of Vivado Integrated Logic Analyzer, especially when you debug your AXI read and write logic in your accelerator design. Check the Appendix C for more detail. Nonetheless, don't rely too much on ILA since it would add extra time to the already long Place-and-Route time. The best way to debug is still writing your own testbenches and simulating it before you run Vivado implementation. A testbench for conv3D has been provided to you. Please feel free to use or change the testbench as you see fit.

```
cd hardware
make iverilog-sim tb=xcel_testbench
```

The testbench does not invoke any software MIF file nor the RISC-V core that you designed in Checkpoint 2. Rather, it tests the functionality of the xcel accelerator (conv3D) with a simple `hardware/sim/mem_model.v` in place. A memory block is used to initialize data for testing. You may want to use the testbench when you design your own conv3D.

Start with something simple that works first, then worry about optimization later. It makes sense to just copying all the weight data and a test image to on-chip BRAM before you start doing the computation. Buffering the temporary result to another local buffer, and only write back to the DDR once everything is done. Using burst mode is highly recommended. As a reference, you can take a look at the code in `hardware/src/accelerator/dma/_controller.v` to see some examples of how to setup a burst read/write request, and then how to convert the AXI interface to the BlockRAM interface and vice versa. You could do something similar to that in your accelerator.

Once you have the data on chip, the low hanging fruit is to do some unrolling or pipelining for parallel execution. Make use of dual-port memory modules to allow you to have more parallel accesses. In addition, try not to use a monolithic memory for everything, but instead split it to smaller modules and assign one for each compute unit.

Please also feel free to add extra parameters to your accelerator. You need to change the IO memory-mapped addresses in software and your Riscv151 IO controller to support the new parameters.

Should you need to make changes to any of the files given to you, either to make your implementation more efficient or functioning, please feel free to do so.

3.11 Checkpoint 3 Deliverables

In-lab Checkoff: Demonstrate that your accelerator implementation `xcel_opt` works correctly with the *lenet* software, i.e. it should produce a result that matches the software implementation when testing with the first 128 images from the MNIST dataset. You should be able to obtain some speedup over the given baseline `xcel_naive` implementation. There is no target speedup for the checkpoint as long as you implement some memory optimization (e.g., buffering) and compute optimization (e.g., loop unrolling, pipelining) for your accelerator.

Additionally, please write a short report (push to *docs/*) with your answers to the following questions

- Sketch the block diagram of your accelerator design. What memory or compute optimizations would you like to highlight in your design?
- What is the speedup over the software execution? What about the baseline accelerator implementation?
- Where do you think the speedup comes from? What makes the software (CPU) execution so inefficient in comparison to specialized hardware architecture?
- The RV32I that our Riscv151 supports does not have a multiply instruction. Therefore, a multiplication is carried out by a software routine instead (look at the **times** function). Can you come up with an approach to implement a multiply operation for your processor (that should be more efficient than invoking **times** function)? You don't have to implement it, just briefly describe how you would do it and how many cycles it would take.

(You should think of an approach that requires the *least impact* to your existing processor pipeline.)

- What is the maximum achievable frequency of your implementation now?

4 Final Checkpoint - Optimization

TBD

5 Grading and Extra Credit

All groups must complete the final checkoff by May 05, 2021. If you are unable to make the deadline for any of the checkpoints, it is still in your best interest to complete the design late, as you can still receive most of the credit if you get a working design by the final checkoff.

5.1 Checkpoints

We have divided the project up into checkpoints so that you (and the staff) can pace your progress.

5.2 Style: Organization, Design

Your code should be modular, well documented, and consistently styled. Projects with incomprehensible code will upset the graders.

5.3 Final Project Report

Upon completing the project, you will be required to submit a report detailing the progress of your EECS151/251A project. The report should document your final circuit at a high level, and describe the design process that led you to your implementation. We expect you to document and justify any tradeoffs you have made throughout the semester, as well as any pitfalls and lessons learned. Additionally, you will document any optimizations made to your system, the system's performance in terms of area (resource use), clock period, and CPI, and other information that sets your project apart from other submissions.

The staff emphasizes the importance of the project report because it is the product you are able to take with you after completing the course. All of your hard work should reflect in the project report. Employers may (and have) ask to examine your EECS151/251A project report during interviews. Put effort into this document and be proud of the results. You may consider the report to be your medal for surviving EECS151/251A.

5.3.1 Report Details

You will turn in your project report on Gradescope by the final checkoff date. The report should be around 8 pages total with around 5 pages of text and 3 pages of figures (\pm a few pages on each). Ideally you should mix the text and figures together.

Here is a suggested outline and page breakdown for your report. You do not need to strictly follow this outline, it is here just to give you an idea of what we will be looking for.

- **Project Functional Description and Design Requirements.** Describe the design objectives of your project. You don't need to go into details about the RISC-V ISA, but you need to describe the high-level design parameters (pipeline structure, memory hierarchy, etc.) for this version of the RISC-V. (\approx 0.5 page)
- **High-level organization.** How is your project broken down into pieces. Block diagram level-description. We are most interested in how you broke the CPU datapath and control down into submodules, since the code for the later checkpoints will be pretty consistent across all groups. Please include an updated block diagram (\approx 1 page).

- **Detailed Description of Sub-pieces.** Describe how your circuits work. Concentrate here on novel or non-standard circuits. Also, focus your attention on the parts of the design that were not supplied to you by the teaching staff. (≈ 2 pages).
- **Status and Results.** What is working and what is not? At what frequency (50MHz or greater) does your design run? Do certain checkpoints work at a higher clock speed while others only run at 50 MHz? Please also provide the area utilization. Also include the CPI and minimum clock period of running `mmult` for the various optimizations you made to your processor. This section is particularly important for non-working designs (to help us assign partial credit). (≈ 1 -2 pages).
- **Conclusions.** What have you learned from this experience? How would you do it different next time? (≈ 0.5 page).
- **Division of Labor. This section is mandatory. Each team member will turn in a separate document from this part only.** The submission for this document will also be on Gradescope. How did you organize yourselves as a team. Exactly who did what? Did both partners contribute equally? Please note your team number next to your name at the top. (≈ 0.5 page).

When we grade your report, we will grade for clarity, organization, and grammar. Submit your report to the Gradescope assignment. Only one partner needs to submit the shared report, while each individual will need to submit the division of labor report to a separate Gradescope assignment.

5.4 Extra Credit

Teams that have completed the base set of requirements are eligible to receive extra credit worth up to 10% of the project grade by adding extra functionality and demonstrating it at the time of the final checkoff.

The following are suggested projects that may or may not be feasible in one week.

- Branch Predictor: Implement a two bit (or more complicated) branch predictor with a branch history table (BHT) to replace the naive 'always taken' predictor used in the project
- 5-Stage Pipeline: Add more pipeline stages and push the clock frequency past 100MHz
- RISC-V M Extension: Extend the processor with a hardware multiplier and divider
- TBD

When the time is right, if you are interested in implementing any of these, see the staff for more details.

5.5 Project Grading

80% Functionality at project due date. You will demonstrate the functionality of your processor during the final interview.

5% Optimization at project due date. This score is contingent on implementing all the required functionality. An incomplete project will receive a zero in this category.

5% Checkpoint functionality. You are graded on functionality for each completed checkpoint. The total of these scores makes up 5% of your project grade. The weight of each checkpoint's score may vary.

10% Final report and style demonstrated throughout the project.

Not included in the above tabulations are point assignments for extra credit as discussed above. Extra credit is discussed below:

Up to 10% Additional functionality. Credit based on additional functionality will be qualified on a case by case basis. Students interested in expanding the functionality of their project must meet with a GSI well ahead of time to be qualified for extra credit. Point value will be decided by the course staff on a case by case basis, and will depend on the complexity of your proposal, the creativity of your idea, and relevance to the material taught.

A Local Development

You can build the project on your laptop but there are a few dependencies to install. In addition to Vivado and Icarus Verilog, you need a RISC-V GCC cross compiler and an `elf2hex` utility.

A.1 Linux

A system package provides the RISC-V GCC toolchain (Ubuntu): `sudo apt install gcc-riscv64-linux-gnu`. There are packages for other distros too.

To install `elf2hex`:

```
git clone git@github.com:sifive/elf2hex.git
cd elf2hex
autoreconf -i
./configure --target=riscv64-linux-gnu
make
vim elf2hex # Edit line 7 to remove 'unknown'
sudo make install
```

A.2 OSX, Windows

Download SiFive's GNU Embedded Toolchain [from here](#). See the 'Prebuilt RISC-V GCC Toolchain and Emulator' section.

After downloading and extracting the tarball, add the `bin` folder to your `PATH`. For Windows, make sure you can execute `riscv64-unknown-elf-gcc -v` in a Cygwin terminal. Do the same for OSX, using the regular terminal.

For Windows, re-run the Cygwin installer and install the packages `git`, `python3`, `python2`, `autoconf`, `automake`, `libtool`. See [this StackOverflow question](#) if you need help selecting the exact packages to install.

Clone the `elf2hex` repo `git clone git@github.com:sifive/elf2hex`. Follow the instructions in the [elf2hex repo README](#) to build it from git. You should be able to run `riscv64-unknown-elf-elf2hex` in a terminal.

B BIOS

This section was written by Vincent Lee, Ian Juch, and Albert Magyar.

B.1 Background

For the first checkpoint we have provided you a BIOS written in C that your processor is instantiated with. BIOS stands for Basic Input/Output System and forms the bare bones of the CPU system on initial boot up. The primary function of the BIOS is to locate, and initialize the system and peripheral devices essential to the PC operation such as memories, hard drives, and the CPU cores.

Once these systems are online, the BIOS locates a boot loader that initializes the operating system loading process and passes control to it. For our project, we do not have to worry about loading the BIOS since the FPGA eliminates that problem for us. Furthermore, we will not deal too much with boot loaders, peripheral initialization, and device drivers as that is beyond the scope of this class. The BIOS for our project will simply allow you to get a taste of how the software and hardware layers come together.

The reason why we instantiate the memory with the BIOS is to avoid the problem of bootstrapping the memory which is required on most computer systems today. Throughout the next few checkpoints we will be adding new memory mapped hardware that our BIOS will interface with. This document is intended to explain the BIOS for checkpoint 1 and how it interfaces with the hardware. In addition, this document will provide you pointers if you wish to modify the BIOS at any point in the project.

B.2 Loading the BIOS

For the first checkpoint, the BIOS is loaded into the Instruction memory when you first build it. As shown in the Checkpoint 1 specification, this is made possible by instantiating your instruction memory to the BIOS file by building the block RAM with the `bios151v3.hex` file. If you want to instantiate a modified BIOS you will have to change this `.hex` file in your block RAM directory and rebuild your design and the memory.

To do this, simply `cd` to the `software/bios151v3` directory and make the `.hex` file by running “make”. This should generate the `.hex` file using the compiler tailored to our ISA. The block RAM will be instantiated with the contents of the `.hex` file. When you get your design to synthesize and program the board, open up screen using the same command from Lab 6:

```
screen $SERIALTTY 115200
```

Once you are in `screen`, if your CPU design is working correctly you should be able to hit Enter and a carrot prompt `'>'` will show up on the screen. If this doesn't work, try hitting the reset button on the FPGA which is the center compass switch and hit enter. If you can't get the BIOS carrot to come up, then your design is not working and you will have to fix it.

B.3 Loading Your Own Programs

The BIOS that we provide you is written so that you can actually load your own programs for testing purposes and benchmarking. Once you instantiate your BIOS block RAM with the `bios151v3.hex` file and synthesize your design, you can transfer your own program files over the serial line.

To load your own programs into the memory, you need to first have the `.hex` file for the program compiled. You can do this by copying the software directory of one of our C programs folders in `/software` directory and editing the files. You can write your own MIPS program by writing test code to the `.s` file or write your own C code by modifying the `.c` file. Once you have the `.hex` file for your program, impact your board with your design and run:

```
hex_to_serial <file name> <target address>
```

The `<file name>` field corresponds to the `.hex` file that you are to uploading to the instruction memory. The `<target address>` field corresponds to the location in memory you want to write

your program to.

Once you have uploaded the file, you can fire up screen and run the command:

```
jal <target hex address>
```

Where the `<target hex address>` is where you stored the location of the hex file over serial. Note that our design does not implement memory protection so try to avoid storing your program over your BIOS memory. Also note that the instruction memory size for the first checkpoint is limited in address size so large programs may fail to load. The `jal` command will change the PC to where your program is stored in the instruction memory.

B.4 The BIOS Program

The BIOS itself is a fairly simple program and composes of a glorified infinite loop that waits for user input. If you open the `bios151v3.c` file, you will see that the main method composes of a large for loop that prints a prompt and gets user input by calling the `read_token` method. If at any time your program execution or BIOS hangs or behaves unexpected, you can hit the reset button on your board to reset the program execution to the main method. The `read_token` method continuously polls the UART for user input from the keyboard until it sees the character specified by `ds`. In the case of the BIOS, the termination character `read_token` is called with is the `0xd` character which corresponds to Enter. The `read_token` method will then return the values that it received from the user. Note that there is no backspace option so if you make a mistake you will have to wait until the next command to fix it.

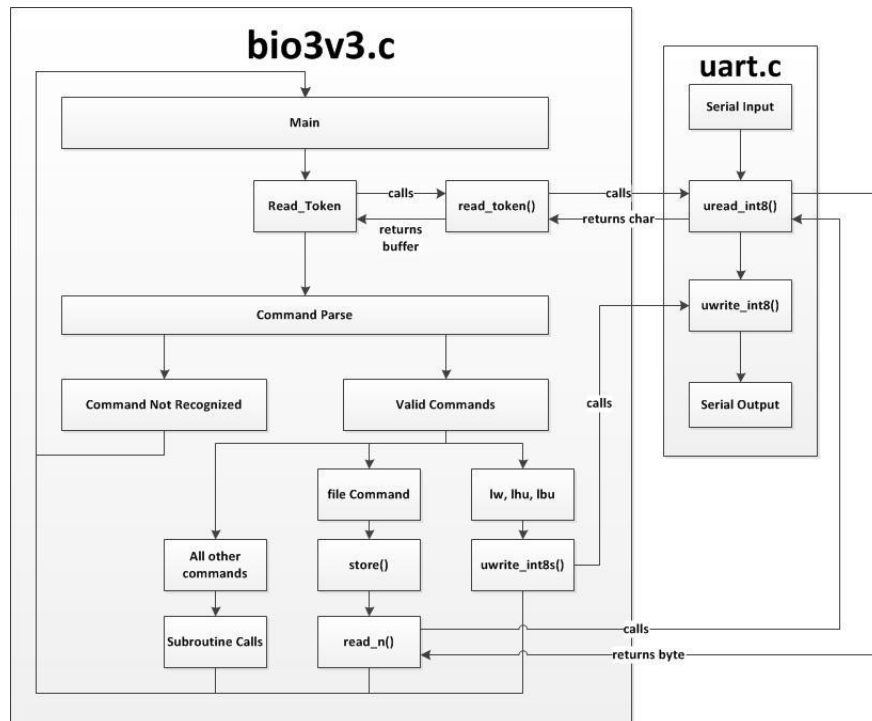


Figure 8: BIOS Execution Flow

The buffer returned from the `read_token` method with the user input is then parsed by comparing the returned buffer against commands that the BIOS recognizes. If the BIOS parses a command successfully it will execute the appropriate subroutine or commands. Otherwise it will tell you that the command you input is not recognized. If you want to add commands to the BIOS at any time in the project, you will have to add to the comparisons that follow after the `read_token` subroutine in the BIOS.

B.5 The UART

You will notice that some of the BIOS execution calls will call subroutines in the `uart.c` file which takes care of the transmission and reception of byte over the serial line. The `uart.c` file contains three subroutines. The first subroutine, `uwrite_int8` executes a UART transmission for a single byte by writing to the output data register. The second subroutine `uwrite_int8s` allows you to process an array of type `int8_t` or `chars` and send them over the serial line. The third routine `uread_int8` polls the UART for valid data and reads a byte from the serial line.

In essence, these three routines are operating the UART on your design from a software view using the memory mapped I/O. Therefore, in order for the software to operate the memory map correctly, the `uart.c` module must store and load from the correct addresses as defined by our memory map. You will find the necessary memory map addresses in the `uart.h` file that conforms to the design specification.

B.6 Command List

The following commands are built into the BIOS that we provide for you. All values are interpreted in hexadecimal and do not require any radix prefix (ex. "0x"). Note that there is not backspace command.

`jal <hexadecimal address>` - Moves program execution to the specified address
`lw <hexadecimal address>` - Displays word at specified address to screen
`lhu <hexadecimal address>` - Displays half at specified address to screen
`lbu <hexadecimal address>` - Displays byte at specified address to screen
`sw <value> <hexadecimal address>` - Stores specified word to address in memory
`sh <value> <hexadecimal address>` - Stores specified half to address in memory
`sb <value> <hexadecimal address>` - Stores specified byte to address in memory

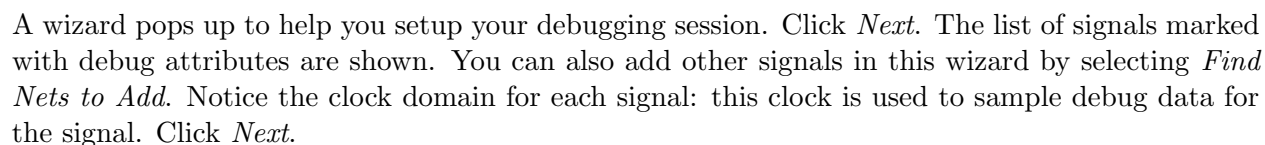
There is another command file in the `main()` method that is used only when you execute `hex_to_serial`. When you execute `hex_to_serial`, your workstation will initiate a byte transfer by calling this command in the BIOS. Therefore, don't mess with this command too much as it is one of the more critical components of your BIOS.

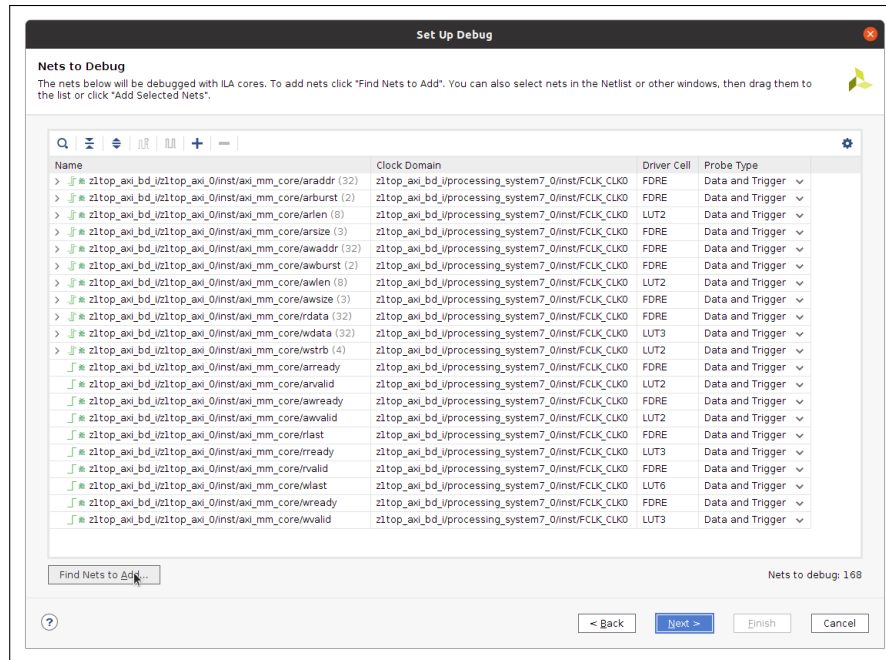
B.7 Adding Your Own Features

Feel free to modify the BIOS code if you want to add your own features during the project for fun or to make your life easier. If you do choose to modify the BIOS, make sure to preserve essential

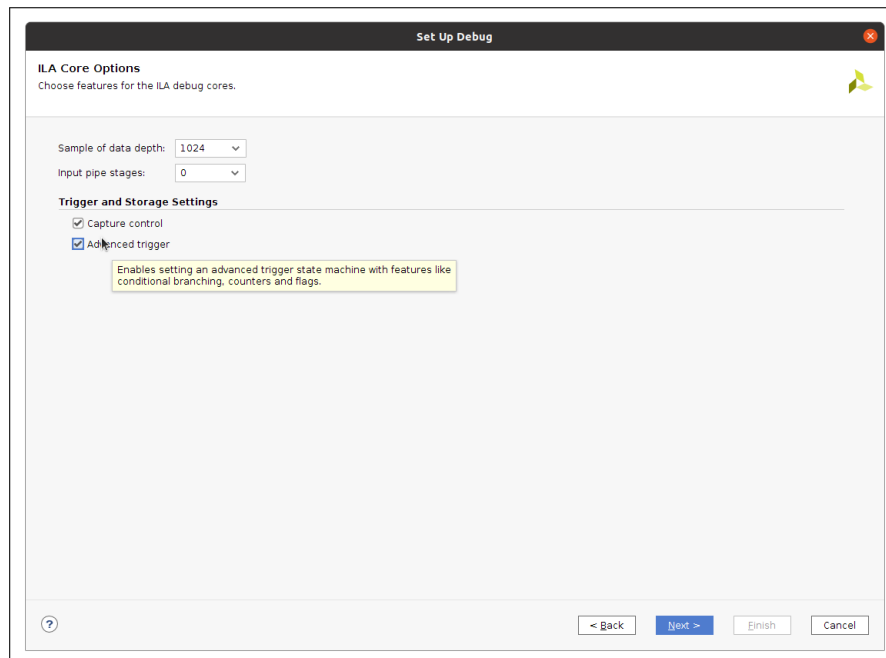
C Debugging with Vivado Integrated Logic Analyzer

Inspect the file `src/accelerator/axi_mm_adapter.v`. Some AXI signals are annotated with `(* mark_debug = "true" *)`. This tells Vivado not to trim or optimize away these signals, and that they can be monitored/probed during debugging session. Open your Vivado project in GUI mode (make sure your project finished the Synthesis step). In the *Flow Navigator* panel, under *Synthesis*, click *Set Up Debug*.

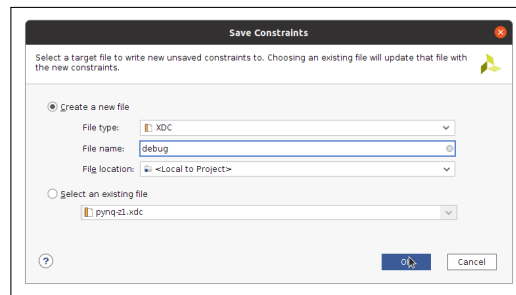




In this wizard, we will set the depth of the captured data (number of debug/captured data items) for our debugged signals. The bigger the depth value, the more debug data (or the longer the runtime) can be captured. However, since a debug core uses Block RAMs to store the captured data, your implementation might run out of on-chip memory storage if you set the depth too big. The debug cores actually consume resources; they are not free, so you should always be mindful of the current resource utilization of your design. In addition, make sure *Capture control* and *Advanced trigger* are checked. Hit *Next*, and finally hit *Finish* on the *Set up Debug Summary* page to close the wizard.



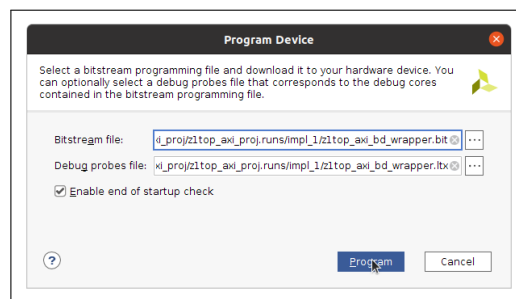
Now, we generate a new bitstream with the debug ILA cores: *Generate Bitstream*. Vivado will ask where we should save the debug constraints. Select *Create a new file* as in the following picture to avoid overwriting the existing constraint file. Click *OK* to finish the debug setup.



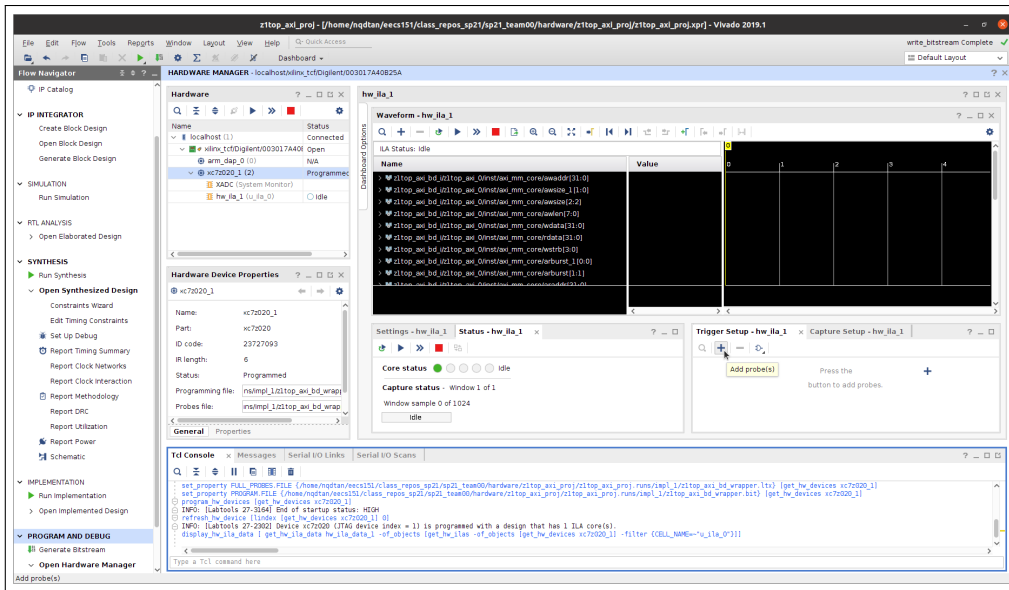
After bitstream generation, we connect and program the FPGA as usual. This time, you will see that, in addition to the bitstream file, we also load the debug probes file. But before doing that, we need to initialize the ARM core (or the Zynq PS) by running the following command

```
make init-arm
```

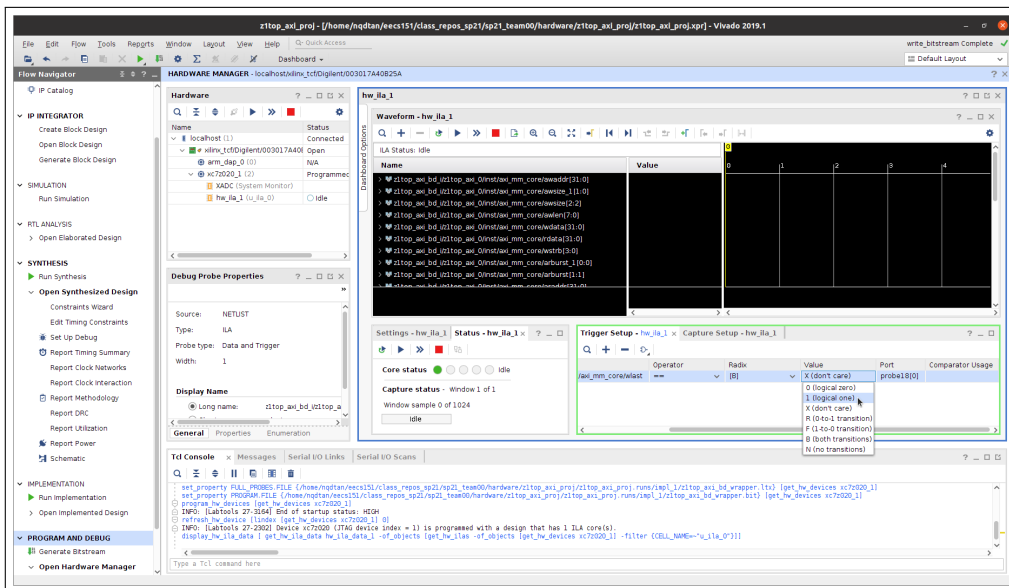
Otherwise, this would not work, since our logic receives the clock signal from the Zynq PS.



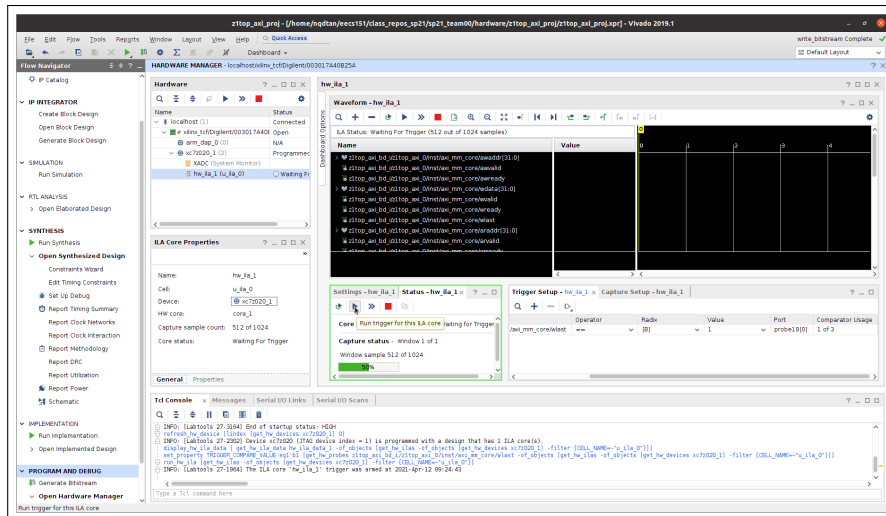
The waveform pane will show up after we program the FPGA with the probes file. This waveform window looks like the Simulation waveform window, but the cool thing is that we can setup trigger event to ignite the transitions of the signals we are concerning with. To do so, in the *Trigger Setup* - *hw_ila_1* pane, click the plus button to add probes.



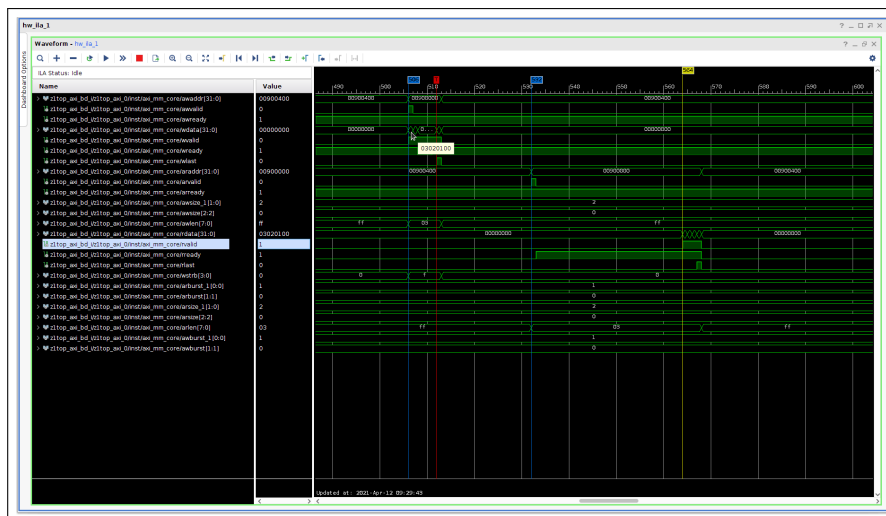
Select the signal `wlast` from the popped-up window. Now, we will set the trigger condition. Also in the *Trigger Setup - hw_ila_1* pane, select *1 (logical one)* for the *Value* column. The remaining columns can be kept unchanged. Essentially, we are comparing `wlast` to 1: if the comparison is true (the last write data of an AXI Write transaction is sent to the bus), this will trigger the ILA cores.



Now, in the *Status - hw_ila_1* pane, click the play button ("Run trigger for this ILA core"). It runs until waiting for trigger (50%). To complete the core execution, we will provide a **real** trigger. To that end, the `software/axi_test` program is used as a demo. Go to `software/axi_test`, do make run to load the assembly instruction of the program to your CPU's Memory blocks. Then open the screen program, and do `jal 10000000` to execute the program.



This will start the DMA execution, which in turn sends the AXI write transaction, and then triggers the ILA core. Now you will see the waveform updated! This is all happening when your PYNQ board is actually running.



Let's take a closer look at the waveform to understand what just happened. The red marker indicates when the trigger condition is met (i.e, `wlast` became HIGH). On the write transaction, we can see a burst write of four 32-bit data items. This matches what is written in the `axi_test` program, as we are sending an array of 16 8-bit items on the 32-bit AXI data bus. On the read transaction, a burst read of four 32-bit data items is received, which confirms our expectation from the software program too. Also notice that latency between the read request fire (valid and ready are both HIGH) and the first read data response fire (from the second blue marker to the yellow marker). This is the off-chip read latency from the DDR to our DMA engine on the FPGA (roughly 32 cycles in this case).



We can also observe that the read data matches the write data.

Address	Data	Status
00000000	00000000	Write
00000000	00000000	Read

Try modifying the axi test program to test different scenarios until you understand how the bus works (e.g., increasing the DMA transfer length, or sending/receiving 32-bit data instead of 8-bit). Hopefully this example demonstrates to you how to use ILA to get meaningful messages in real time to debug your implementation.