NYU-6463-RV32I Processor Design Project

(Version 1 Specification)

Groups of 2 or 3 (no difference in difficulty!). Project Deadlines: Nov 4, Nov 18, Dec 16, Dec 20.

30 Points

For the final project, you will implement a 32-bit processor in VHDL or Verilog, called NYU-6463-RV32I Processor. It will be capable of executing arbitrary programs and run on your FPGA hardware.

1. Design Specification

The NYU-6463-RV32I processor is a 32-bit architecture which executes a subset of the open source RISC-V RV32I instruction set. There are three main instruction types: (a) computational operations (from register file to register file), (b) load/store between memory and register file, and (c) control flow (jumps and branches to different parts of code). The instruction formats and instruction set are detailed in the **Supplementary Material Part 1**. You can also find details on the full specification at https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

2. Processor Components

The processor comprises the following components. An example datapath is presented in Fig. 1 (on the next page).

- **Program counter (PC) register:** This is a 32-bit register that contains the address of the next instruction to be executed by the processor. Upon reset this should equal the start address of instruction memory (0x01000000).
- **Control Unit:** This block takes as input some or all of the 32 bits of the instruction, and computes the proper control signals that are required for correctly coordinating the other blocks in your design. These signals are generated based on the type and the content of the instruction being executed. This will contain a FSM.
- **Register File:** This block contains 32 32-bit registers. The register file supports two independent register reads and one register write in one clock cycle. 5 bits are used to address each register.
    - E.g. R3 = R1 + R2
    - Note: R0 is special, and is a read-only register that is always hardwired to equal zero.
- **ALU:** This block performs operations such as addition, subtraction, comparison, etc. It uses the control signals generated by the Decode Unit, as well as the data from the registers or from the instruction directly. It computes data that can be written into one of the registers (including PC). You will implement this block by referring to the instruction set.
- **Instruction and Data Memory:** The instruction memory is initialized to contain the program to be executed. Instruction memory width is 4 bytes (32-bits), although it is byte-addressed. The data memory stores the data and is accessed using `LW` (load word) and `SW` (store word) instructions. Data memory width is 4-bytes (32-bits), although it is also byte-addressed and supports read/write to individual bytes.
    - Instruction and Data Memory accesses are restricted to 4-byte alignment in the NYU-6463-RV32I.
    - Your instruction memory should be at least 2KBytes in size. It shall begin at address 0x01000000.
        - Hint: You will likely wish to **infer Block RAMs** for architecting the ROM efficiently in your design. In order to perform this your Verilog instruction memory will likely need to be *word*-indexed and 32-bits wide. This means you will need to perform some address translation to convert the byte address indexes to your implemented word address indexes. See Supplementary Material Part 2 (the assembly guide), Part B, for further proposed implementation details.
        - Upon reset, your PC should be set to 0x01000000.
    - Your data memory should be at least 4KBytes in size. It shall begin at address 0x80000000.

- Hint: Your data memory can be built either using a single long 32-bit memory array or using 4 interleaved sets of 8-bit (one-byte) wide memories. Again, you will likely **not** be able to successfully construct it using one long 8-bit wide memory array, as this will most likely not successfully infer as block-RAM. As such, manage your memory addresses and decoding carefully when constructing your data memory module.

- You must implement special read-only memory-mapped values at addresses 0x00100000, 0x00100004, 0x00100008 that when read returns the N number(s) of the members of your group.
- (Optional) If you choose to also run your design on the real Basys3/Nexys4 FPGA development boards, you may also implement read-only switches at address 0x00100010 and read/write LEDs at address 0x00100014.

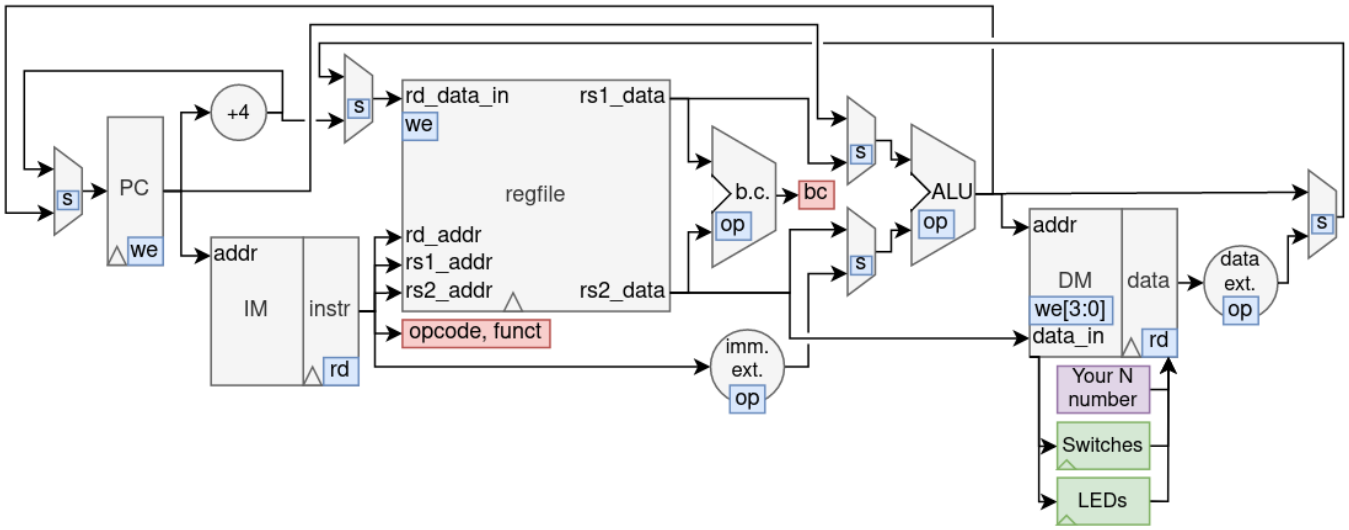

Fig. 1: NYU-6463-RV32I Processor. Suggested control signals (from FSM) in blue, status signals (to FSM) in red.
(Note: b.c. stands for "Branch comparisons". Your control/status signals might be different. This datapath is for illustrative purposes only. This schematic is by no means final, exhaustive, or even necessarily correct — it is just a conceptual representation of how the components could go together. You can add control/status/data signals and other components as you require with justification. The green boxes for Switches and LEDs are required and will be connected to the relevant components on the Basys3/Nexys4 FPGA development boards).

3. Processor Operation

The NYU-6463-RV32I Processor performs the tasks of instruction fetch, instruction decode, execution, memory access and write-back in a multi-cycle manner.

- Firstly, the PC value is used as an address to index the instruction memory which supplies a 32-bit value of the next instruction to be executed.
- This instruction is then divided into the different fields shown in Supplementary Material, Table 1. The instructions' opcode and function field bits are sent to the decode FSM unit to determine the type of instruction to execute.
- The type of instruction then determines which control signals are to be asserted and what function the ALU and other components will perform.
- Certain datapath signals and values can be derived directly from the instruction.
  - For example, the instruction register address fields rs2 bits [24 - 20], rs1 bits[19 - 15], and rd bits [11 - 7] are used to address the register file.
- Each instruction may take a varying number of clock cycles to execute as the FSM invokes the different parts of the Datapath.

Depending upon the given instruction, register and memory values may be read or written. Data values can also be operated on by the ALU. Each individual operation is determined by the control unit to either compute a memory address (e.g. load or store), perform an arithmetic operation (e.g. and, or, xor and add, xor and sub), or compare (e.g. branch). If the instruction decoded is arithmetic, the ALU result must be written to a register. If the instruction decoded is a load or a store, the ALU result is used to address the data memory. The final step writes the ALU result or memory value back to the register file.

4. Task Overview

- Implement the multi-cycle NYU-6463-RV32I Processor as per the specification. You may use either Verilog or VHDL for any file in the project. NOTE: You need to implement only the instructions described in Supplementary Material Table 3/4. You cannot add additional instructions.

- Do a performance (max. speed of your processor) and area (number of gates you used from each type) analysis of your design. Explain your analysis comprehensively. You may be expected to identify and explain the critical path of your design. **Your design must be synthesizable. It must be able to interface with the switches and LEDs.**

    - You may wish to refer to the Xilinx Vivado guide on synthesizing memories: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf

    - (Optional) You may also connect your design to a UART peripheral. This will be worth bonus marks / extra credit.

- Your design should be simulatable (**both** functional simulation and timing simulation). Write program(s)/test(s) to thoroughly test your processor design.

    - Your aim is to stress-test **all** parts of your design. You can think of this as though you are checking that all the lines of your RTL code are simulated and work as intended, or you can think of this as checking that all the wires, flip-flops, and gates in your design are operating correctly.

    - You need to perform both low-level tests (e.g. on individual components and instructions) and high-level tests (e.g. complex algorithms stored in instruction memory that the CPU executes).

        - Required: You should have testbenches for every individual component in your design (unit tests) *and* testbenches for the entire assembled processor (integration tests).

        - Required: At least two high-level complex programs for the entire processor should be included. Other complex tests may be worth additional bonus marks / extra credit.

            - Required - Test A: Your program should interface with the switches and LEDs, implementing some complex function indicating correct behavior (not RC5).

            - Required - Test B: This should perform RC5 encryption and decryption (you may provide the skey array - key derivation is not required). Ensure it is possible to determine correct behavior using switches and LEDs (or UART, if you used this).

        - (Optional) If your design is robust, you may alternatively write your high-level tests (including RC5) in C and use the riscv32-unknown-elf with RV32I settings to compile it to assembly, then validate those programs.

            - **Note: Support for C will result in bonus marks / extra credit.**

    - Run (simulate + real-world) your program(s) on your designed processor and show that it works properly.

- For each low-level and high-level test case, describe how many clock cycles are required to execute your program(s) on your NYU-6463-RV32I Processor.

5. Final Deliverables:

1. Put your processor source code and assembly code in a zipped folder.
2. Your report in PDF format, including:
    a. Your complete design datapath (e.g. your version of Fig 1.)
        i. 1 mark
    b. Your FSM diagram for multi-cycle control.
        i. 1 mark
    c. A full justification on the correctness of your design, including simulation screenshots and explanations of low-level test cases (and how many clock cycles each test case takes and why)
        i. 2 marks
    d. A performance and area analysis of design
        i. 1 mark
    e. A description of your high-level test-cases in assembly (or C) (and how many clock cycles each test case takes and why) and a demonstration of their output (simulations, screenshots, and videos of it working on the real hardware with audible explanation)
        i. 1 mark per each, if no output give 0.5 marks per each
    f. An explanation of how one could optimize and improve your design in the future. Provide an approximation of how much gain you might achieve and at what cost.
        i. 1 mark
3. A video presentation of your design which covers the above material (~10 minutes)
    a. Recommended: use OBS video capture software for recording.
    b. (We suggest accompanying your presentation with slides)
    c. Structure this presentation as if you were presenting your project as a completed work assignment to your client/boss - problem, solution, why your solution is outstanding.
        i. 2 marks

6. Milestones:

Task	**Due Date**
T1: Prepare a plan and discuss preliminary work with a TA in a 6-minute presentation. Your presentation should include: ● A high-level overview of your processor implementation ● A timeline of implementation tasks and which team-members are responsible ● A plan for testing (how will you know it is working?) ● Mitigation plan if "something goes wrong" (team-members sick etc., task harder than expected). **(3 points)**	**Week of Nov 4**
T2: Develop all individual processor components and corresponding testbenches/simulation scripts **(9 points)**	Nov 18
T3: Interconnect all individual processor components into the complete processor and develop corresponding testbench/simulation **(9 points)**	Dec 16
T4: Complete all remaining documentation tasks (reports and the 10 minute video) **(9 points)**	Dec 20

Supplementary Material Part 1: ISA

Note: This processor implements a subset of the RISC-V RV32I instruction set as defined in the official specification at https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

You may also find the following guide on RISC-V assembly programming useful:
https://shakti.org.in/docs/risc-v-asm-manual.pdf

Table 1: NYU-6463-RV32I Processor instruction types

Bit	*31*	*25, 24*	*20, 19*	*15, 14*	*12, 11*	*7, 6* *z* *0*
R-type	funct7 (7 bits)	rs2 (5 bits)	rs1 (5 bits)	funct3 (3 bits)	rd (5 bits)	Opcode (7 bits)
I-type	imm[11:0]		rs1	funct3	rd	Opcode
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	Opcode
B-type	imm[12, 10:5]	rs2	rs1	funct3	imm[4:1, 11]	Opcode
U-type	imm[31:12]				rd	Opcode
J-type	imm[20, 10:1, 11, 19:12]				rd	Opcode

Table 2. NYU-6463-RV32I Processor instruction fields

Field	**Description**
funct7, funct3, Opcode	Detail the specific instruction that is being executed
rs2	5-bit specifier for source register 2
rs1	5-bit specifier for source register 1
rd	5-bit specifier for the destination register
imm	Signed immediate used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement

Supported instruction set:

Table 3. NYU-6463-RV32I Processor Mnemonics and Details

Mnemonic	Full name	RTL pseudocode	Details
LUI	Load Upper Immediate	rd={imm[31:12], 12'b0}	Loads the immediate value into the upper 20 bits of the target register rd and sets the lower bits to 0
AUIPC	Add Upper Immediate to PC	rd=PC+{imm[31:12], 12'b0}	Forms a 32-bit offset from the 20-bit value by filling the lower bits with zeros, adds this to PC, and stores the result in rd.
JAL	Jump and Link	rd=PC+4; PC=PC+sign_ext(imm)	Jump to PC=PC+(sign-extended immediate value) and store the current PC

			address+4 in register rd.
JALR	Jump and Link Register	rd=PC+4; PC=rs1+sign_ext(imm)	Jump to PC=rs1 register value +(sign-extended immediate value) and store the current PC address+4 in register rd
BEQ	Branch if Equal	PC=(rs1==rs2) ? PC+sign_ext(imm) : PC+4	Take the branch (PC=PC+(sign-extended immediate value) if rs1 is equal to rs2
BNE	Branch if Not Equal	PC=(rs1!=rs2) ? PC+sign_ext(imm) : PC+4	Take the branch (PC=PC+(sign-extended immediate value) if rs1 is not equal to rs2
BLT	Branch if Less Than (signed)	PC=(signed(rs1)<signed(rs2)) ? PC+sign_ext(imm) : PC+4	Take the branch (PC=PC+(sign-extended immediate value) if signed rs1 is less than signed rs2, otherwise PC=PC+4
BGE	Branch if Greater Than or Equal (signed)	PC=(signed(rs1)>=signed(rs2)) ? PC+sign_ext(imm) : PC+4	Take the branch (PC=PC+(sign-extended immediate value) if signed rs1 is greater than or equal to signed rs2, otherwise PC=PC+4
BLTU	Branch if Less Than (unsigned)	PC=(unsigned(rs1)<unsigned(rs2)) ? PC+sign_ext(imm) : PC+4	Take the branch (PC=PC+(sign-extended immediate value) if unsigned rs1 is less than unsigned rs2, otherwise PC=PC+4
BGEU	Branch if Greater Than or Equal (unsigned)	PC=(unsigned(rs1)>=unsigned(rs2)) ? PC+sign_ext(imm) : PC+4	Take the branch (PC=PC+(sign-extended immediate value) if unsigned rs1 is greater than or equal to unsigned rs2, otherwise PC=PC+4
LB	Load Byte (signed)	rd=sign_ext(data[rs1+sign_ext(imm)][7:0])	Load 8-bit value at memory address [rs1 value]+(sign extended immediate) and store it at rd as a 32-bit sign extended value
LH	Load Half-Word (2 bytes) (signed)	rd=sign_ext(data[rs1+sign_ext(imm)][15:0])	Load 16-bit value at memory address [rs1 value]+(sign extended immediate) and store it at rd as a 32-bit sign extended value
LW	Load Word (4 bytes)	rd=data[rs1+sign_ext(imm)][31:0]	Load 32-bit value at memory address [rs1 value]+(sign extended immediate) and store it at rd
LBU	Load Byte (unsigned)	rd=zero_ext(data[rs1+sign_ext(imm)][7:0])	Load 8-bit value at memory address [rs1 value]+(sign extended immediate) and store it at rd as a 32-bit zero extended value
LHU	Load Half-Word (2 bytes) (unsigned)	rd=zero_ext(data[rs1+sign_ext(imm)][15:0])	Load 16-bit value at memory address [rs1 value]+(sign extended immediate) and store it at rd as a 32-bit zero extended value
SB	Store Byte	data[rs1+sign_ext(imm)][7:0] = rs2[7:0]	Store the lower 8-bits of rs2 to memory

			address [rs1 value]+(sign extended immediate)
SH	Store Half-Word (2 bytes)	data[rs1+sign_ext(imm)][15:0] = rs2[15:0]	Store the lower 16-bits of rs2 to memory address [rs1 value]+(sign extended immediate)
SW	Store Word (4 bytes)	data[rs1+sign_ext(imm)][31:0] = rs2	Store the 32-bits of rs2 to memory address [rs1 value]+(sign extended immediate)
ADDI	Add Immediate	rd = rs1 + sign_ext(imm)	Add the sign-extended immediate to register rs1 and store in rd. Overflow bits ignored.
SLTI	Set Less Than Immediate (signed)	rd = (signed(rs1) < sign_ext(imm)) ? 1 : 0	If signed register rs1 is less than sign-extended immediate value, set register rd to 1, else 0.
SLTIU	Set Less Than Immediate (unsigned)	rd = (unsigned(rs1) < unsigned(sign_ext(imm))) ? 1 : 0	If unsigned register rs1 is less than unsigned (after sign-extending) immediate value, set register rd to 1, else 0.
XORI	XOR with immediate	rd = rs1 ^ sign_ext(imm)	Perform logical XOR operation over rs1 and the sign-extended immediate and place result in register rd.
ORI	OR with immediate	rd = rs1 | sign_ext(imm)	Perform logical OR operation over rs1 and the sign-extended immediate and place result in register rd.
ANDI	AND with immediate	rd = rs1 & sign_ext(imm)	Perform logical AND operation over rs1 and the sign-extended immediate and place result in register rd.
SLLI	Shift Left Logical Immediate	rd = rs1 << imm	Left shift register rs1 value by the immediate value and place result in register rd (shift in zeros).
SRLI	Shift Right Logical Immediate	rd = unsigned(rs1) >> imm	Right shift register rs1 value by the immediate value and place result in register rd (shift in zeros).
SRAI	Shift Right Arithmetic Immediate	rd = signed(rs1) >> imm	Right shift register rs1 value by the immediate value and place result in register rd (shift in the original sign bit).
ADD	Add	rd = rs1 + rs2	Perform the addition of rs1 + rs2 and store in register rd.
SUB	Subtract	rd = rs1 - rs2	Perform the subtraction rs1 - rs2 and store in register rd.
SLL	Shift Left Logical	rd = rs1 << rs2[4:0]	Left shift register rs1 value by the bottom 5 bits of the value in rs2 and place result in register rd (shift in zeros).
SLT	Set Less	rd = (signed(rs1) < signed(rs2)) ? 1 : 0	If signed register rs1 is less than signed

	Than (signed)		rs2, set register rd to 1, else 0.
SLTU	Set Less Than (unsigned)	rd = (unsigned(rs1) < unsigned(rs2)) ? 1 : 0	If unsigned register rs2 is less than unsigned rs2, set register rd to 1, else 0.
XOR	XOR	rd = rs1 ^ rs2	Perform rs1 XOR rs2 and store in register rd.
SRL	Shift Right Logical	rd = rs1 << rs2[5:0]	Right shift register rs1 value by the bottom 5 bits of the value in rs2 and place result in register rd (shift in zeros).
SRA	Shift Right Arithmetic	rd = signed(rs1) << rs2[5:0]	Right shift register rs1 value by the bottom 5 bits of the value in rs2 and place result in register rd (shift in the original sign bit).
OR	OR	rd = rs1 | rs2	Perform rs1 OR rs2 and store in register rd.
AND	AND	rd = rs1 & rs2	Perform rs1 AND rs2 and store in register rd.
FENCE	*Memory FENCE instruction*	*Implement as NOP (addi R0, R0, 0)*	
ECALL	*ECALL service request*	*Implement as HALT (stop program execution)*	
EBREAK	*EBREAK service request*	*Implement as HALT (stop program execution)*	

Encoding for each instruction:

Table 4. NYU-6463-RV32I Processor Instruction Encodings

	Bit fields						
Mnemonic	31:27	26:25	24:20	19:15	14:12	11:7	6:0
LUI	imm[31:12]					rd	0110111
AUIPC	imm[31:12]					rd	0010111
JAL	imm[20|10:1|11|19:12]					rd	1101111
JALR	imm[11:0]			rs1	000	rd	1100111
BEQ	imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011
BNE	imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011
BLT	imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011
BGE	imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011
BLTU	imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011
BGEU	imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011

LB	imm[11:0]		rs1	000	rd	0000011
LH	imm[11:0]		rs1	001	rd	0000011
LW	imm[11:0]		rs1	010	rd	0000011
LBU	imm[11:0]		rs1	100	rd	0000011
LHU	imm[11:0]		rs1	101	rd	0000011
SB	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011
SH	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011
SW	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
ADDI	imm[11:0]		rs1	000	rd	0010011
SLTI	imm[11:0]		rs1	010	rd	0010011
SLTIU	imm[11:0]		rs1	011	rd	0010011
XORI	imm[11:0]		rs1	100	rd	0010011
ORI	imm[11:0]		rs1	110	rd	0010011
ANDI	imm[11:0]		rs1	111	rd	0010011
SLLI	0000000	shamt	rs1	001	rd	0010011
SRLI	0000000	shamt	rs1	101	rd	0010011
SRAI	0100000	shamt	rs1	101	rd	0010011
ADD	0000000	rs2	rs1	000	rd	0110011
SUB	0100000	rs2	rs1	000	rd	0110011
SLL	0000000	rs2	rs1	001	rd	0110011
SLT	0000000	rs2	rs1	010	rd	0110011
SLTU	0000000	rs2	rs1	011	rd	0110011
XOR	0000000	rs2	rs1	100	rd	0110011
SRL	0000000	rs2	rs1	101	rd	0110011
SRA	0100000	rs2	rs1	101	rd	0110011
OR	0000000	rs2	rs1	110	rd	0110011
AND	0000000	rs2	rs1	111	rd	0110011
FENCE	fm	pred succ	rs1	000	rd	0001111
ECALL	000000000000		00000	000	00000	1110011
EBREAK	000000000001		00000	000	00000	1110011