CSSE4010 - Digital System Design - Project Description

School of EECS, University of Queensland, Semester 2 2024
Due date: Friday, 25 October 2024 4PM AEST

FPGA Emulation of Application Specific Instruction Processor (ASIP) Data Path

This project explores FPGA emulation of area and time optimised hardware architectures for carrying out specific arithmetic operations in the context of application specific instruction processors (ASIPs) [1, 2]. Application specific instruction processors (ASIPs) employ tailored instruction set architecture (ISA) for a specific application, implemented via optimised hardware architectures for underlying arithmetic and logic unit and control unit. Such ASIPs offer advantages over general purpose CPUs where tailored hardware and ISA design can lead to better speed, area, and power consumption performance for a given application, and typically, benefit from hardware acceleration and parallel processing.

In this project, you are required to design, implement and test the data path section of an ASIP, emulated on the Nexys 4 FPGA board via a given set of assembly opcodes. In particular, you are required to design specific arithemtic hardware architectures for a given set of assembly instructions. To scale down the problem, the design of a control unit is excluded from this project specification and the selection of assembly instructions is provided through a matrix keypad (will be provided to you). The following specifications apply to the proposed emulation system:

- The assembly instructions to be emulated take the form `OPCODE Rz, Rx, Ry`, where
  - `Rx,Ry` are 8-bit source operands, emulated via the slide switches on the Nexys4 board. i.e. the content of `Rx` is emulated via SW0-SW7 and the content of `Ry` is emulated via SW8-SW15 on the Nexys4 board. Some instructions to be emulated might have only one source operand.
  - `Rz` is the destination operand (i.e. the result) which, with suitable conversions, is to be displayed on the seven segment displays (SSDs) on the Nexys4 board. The size of the result `Rz` might be different for different opcodes to be emulated.
  - `OPCODE` indicates the operation to be performed.
  - There are six different instructions to be emulated.
- The instruction to be emulated is selected via a 4×4 matrix keypad which is interfaced to the Nexys4 board via one of the PMOD connectors. A VHDL module implementing the keypad interfacing and row/column scanning to identify the key being pressed is provided to you. The provided VHDL module also contains the relevant SSD driver, displaying the pressed key on the SSD.
- When a key is pressed on the keypad, the corresponding instruction must be executed by the dedicated arithmetic data path you have designed (with operands provided by the slide switches) and the relevant output must be displayed on SSDs. The list of instructions to be emulated is given below in this document and you are supposed to design, implement and test custom hardware architectures for each opcode with the given design constraints outline in this document. You can assume that keypad inputs are not triggered while an instruction is being emulated (i.e., it is not essential to queue the key presses.)
- A RESET button (the center push button BTNC on the board) should be used which resets the emulation system to a starting state where your 8-digit student number is displayed on SSDs.
- The instructions to be emulated along with any implementation constraints are outlined below. The number on the list indicates the corresponding key on the keypad.
- For each operation to be emulated, functional simulation can be reported in isolation. That is, you can first design, implement and test individual opcodes and subsequently integrate them together with the keypad for opcode selection.

1. `ADDSAT Rz, Rx, Ry`
   (a) **Description:** Fixed-point signed binary addition/subtraction with 8-bit two's complement operands `Rx,Ry` and 8-bit two's complement output `Rz = Rx +Ry`, with output saturation to minimise error due to overflow.
   (b) **Design constraints: Fully behavioural VHDL must not be used**. Implementation must employ a ripple carry adder architecture with necessary logic to detect two's complement overflow and implement output saturation. Top-down design approach must be followed with structural abstraction at the top-level and behavioural/dataflow abstraction for sub-systems.
   (c) **Expected functional simulation:** At least one test scenario to cover positive result with and without overflow and negative result with and without overflow. Whenever there is overflow, the output must be saturated to either `0x80` or `0x7F` appropriately.
   (d) **Expected on-board testing:** The output must be displayed on the SSDs in decimal format with a negative sign when the result is negative. You will need to implement the necessary two's complement to binary coded decimal (BCD) conversion, for which behavioural descriptions are allowed.
   (e) **Hints:** Two's complement 4-bit adder with saturation has been discussed in lectures and a block diagram can be found in lecture notes. This can be extended to 8-bits as required.
2. `INTDIV Rz, Rx, Ry`
   (a) **Description:** Fixed-point integer division with 8-bit unsigned operands `Rx,Ry` and 8-bit unsigned output `Rz = Rx/Ry`, which gives the quotient of the integer division operation. The remainder of the division operation need not to be shown and `Ry`$\neq 0$ condition can be assumed.
   (b) **Design constraints: Fully behavioural VHDL must not be used**. Implementation must follow a serial-shift and parallel-subtraction approach based on the repeated subtraction method. Top-down design approach must be followed with structural abstraction at the top-level and behavioural/dataflow abstraction for sub-systems. The use of datapath and controller approach is suggested.
   (c) **Expected functional simulation:** At least two test cases when the remainder is zero and two cases when the remainder is non-zero.
   (d) **Expected on-board testing:** The quotient is to be displayed on on SSDs in decimal (i.e. BCD) format with three digits for the same test cases used for functional simulation.
   (e) **Hints:** An architecture for a 4-bit shift-subtract divider has been presented in the tutorial class which can be extended for the required 8-bit design.
3. `INTSQRT Rz, RX`
   (a) **Description:** Fixed-point integer square root calculation of 8-bit unsigned operand `Rx`. E.g., $Rx = 4 \Rightarrow Rz = 2$, $Rx = 12 \Rightarrow Rz = 3$ and so on. One algorithm to compute integer square root calculation is provided in the hint.
   (b) **Design constraints:** A data path and controller design approach must be employed and behavioural descriptions are allowed to describe relevant components within the data path and controller state machine within the design. A Top-down design approach must be followed. Xilinx/AMD IP blocks such as CORDIC must not be used.
   (c) **Expected functional simulation:** At least four simulation test cases confirming the correct computation of integer square root for $0 \leq Rx < 50$, $50 \leq Rx < 100$, $100 \leq Rx < 200$, $200 \leq Rx < 256$.
   (d) **Expected on-board testing:** The output must be displayed on two SSDs in decimal (BCD) format for the same test cases used for functional simulation.
   (e) **Hints:** The integer part of the square root of `Rx` can be obtained by repeated subtraction of odd integers from `Rx` and counting how many times such a subtraction can be performed without the answer being negative. Thus obtained count is the integer part of the square root of `Rx`. The previous design of repeated subtraction for division can be extended to implement this functionality.

4. `EUCGCD Rz, Rx, Ry`
   (a) **Description:** The 8-bit unsigned output `Rz` is the greatest common divisor (GCD) of the two 8-bit unsigned inputs `Rx,Ry`. The GCD can be computed using the Euclidean algorithm [3] which uses repeated comparisons and subtraction, as outlined in the hint section.
   (b) **Design constraints:** A data path and controller design approach must be employed and behavioural descriptions are allowed to describe relevant components within the data path and controller state machine within the design. A Top-down design approach must be followed and any Xilinx/AMD IP blocks must not be used.
   (c) **Expected functional simulation:** At least four simulation test cases showing correct computation of GCD, including one scenario where the inputs are co-prime.
   (d) **Expected on-board testing:** The output must be displayed on two SSDs in decimal (BCD) format for the same test cases used for functional simulation.
   (e) **Hints:** The GCD can be computed using the Euclidean algorithm [3] based on iterative computation of absolute difference between the two numbers. Some example FPGA based implementations are presented in [4, 5].
5. `MFMUL Rz, Rx, Ry`
   (a) **Description:** Minifloat multiplication using 8-bit minifloat operands `Rx,Ry` and 8-bit output `Rz = Rx*Ry` also in minifloat format. The floating point number representation minifloat employs 8 bits to represent numbers according to IEEE 754 floating point standard albeit at reduced word length [6]. The minifloat format uses one sign bit, 4 biased exponent bits (with bias 7, i.e. the exponent is stored in excess-7 format) and 3 mantissa bits, providing greater accurancy and range compared with equivalent fixed-point formats.
   (b) **Design constraints:** A top-down design approach must be employed with clearly defined subsystems to perform the required elementary operations within the floating point multiplication. You may have to do some additional reading on floating point arithmetic operations and one example reference is provided in the hint section. For simplicity, you can exclude treatments for special formats (INF, NAN and de-normalised numbers) in your design and it is sufficient to consider only the regular numbers.
   (c) **Expected functional simulation:** At least four test cases showing the correct minifloat multiplication. The simulation output can be directly shown in 8-bit minifloat representation (or equivalent hex format).
   (d) **Expected on-board testing:** The same test cases as the functional simulation. The 8 bits of the output should be displayed on the 8 SSDs with decimal point activated on relevant digits to separate the sign bit, exponent and mantissa fields. Sign bit must be shown on the left most SSD.
   (e) **Hints:** Required steps for floating point multiplication and relevant hardware architecture are given in [7, Section 4] for 32-bit floating point numbers, which can be adopted for minifloat representation.
6. `MFADD Rz, RX, Ry`
   (a) **Description:** Minifloat addition using 8-bit minifloat operands `Rx,Ry` and 8-bit output `Rz=Rx+Ry` also in minifloat format.
   (b) **Design constraints:** A top-down design approach must be employed with clearly defined subsystems to perform the required elementary operations within the floating point addition. You may have to do some additional reading on floating point arithmetic operations and one example reference is provided in the hint section. For simplicity, you can exclude treatments for special formats (INF, NAN and de-normalised numbers) in your design and it is sufficient to consider only the regular numbers.
   (c) **Expected functional simulation:** At least four test cases showing the correct minifloat addition. The simulation output can be directly shown in 8-bit minifloat representation (or equivalent hex format).
   (d) **Expected on-board testing:** The same test cases as the functional simulation. The 8 bits of the output should be displayed on the 8 SSDs with decimal point activated on relevant digits to separate the sign bit, exponent and mantissa fields. Sign bit must be shown on the left most SSD.
   (e) **Hints:** Required steps for floating point addition and relevant hardware architecture are given in [7, Section 3] for 32-bit floating point numbers, which can be adopted for minifloat representation.

Submission

You are required to submit the following items as two separate submissions on Blackboard by the due date **(Friday 25 October 2024 4PM AEST)**:

1. A zip file containing your Vivado project, design source VHDL files, simulation source VHDL files, constraint file and a readme file identifying the top level entity. There is no need to include any of the files generated after synthesis and implementation. **Your submitted files will be simulated, synthesised and tested under Xilinx/AMD Vivado 2022.1/2 and you must make sure compatibility by testing on lab computers before submission.** This will be a Blackboard submission
2. An electronically typeset PDF report containing the items outlined below - this will be a Turnitin submission on Blackboard.

Late penalties apply as per the ECP unless you have an approved extension. This project is worth 20% of the final course marks.

Project Report Content

The following sections are suggested for the report with an indicative page limit. The page limit is an indication only to give you an idea about the depth of information expected.

- Introduction and overall design: Provide a brief description to your overall design, any limitations and assumptions made. It is suggested to employ an overall diagram to capture your design, which you can expand in subsequent sections. [not more than 1 page].
- Functional simulation and evidence of on-board testing for the reset operation showing your 8-digit student number on the SSDs. For functional simulation, SSD multiplexing need not to be shown. [not more than 0.75 page]
- A separate section for emulation of each opcode 1-6 mentioned above. If you haven't implemented an opcode at all then you don't need to add a section for that. If you have partially implemented an opcode, you can still include that and explain your design to receive partial marks. Each section for an opcode should be not more than 2 pages long and should include:
  - Brief description of the design with a diagram (which can be hand-drawn and scanned or typeset). This diagram can be expanding a block in your overall block diagram from the introduction.
  - Functional simulation output from Vivado. This can be the functional simulation output in isolation for the particular opcode being simulated. That is, there no need to include simulation for keypad interfacing or SSD multiplexing. Having `Rx,Ry` as inputs and `Rz` as the output (displayed in suitable form such as binary or hex as appropriate) is sufficient for the functional simulation.
  - Evidence of on-board testing. Some pictures showing on-board testing showing the slide switches and SSDs with the required number of minimum test cases per each opcode.
- FPGA resources and timing performance for the overall design [not more than 0.75 page]
- A conclusions section [not more than 0.5 page]

Marking Criteria

Mark distribution is shown below with partial marks indicated for each operation. Marks will be awarded based on the report content as well as testing your submitted files on board. The project will be marked out of 40 which contributes 20% towards your final course marks.

	Design description with a diagram	Functional Sim	Board Testing	Total
Introduction and overal design description				2
RESET operation		1	1	2
ADDSAT	1	1	1	3
INTDIV	2	1	1	4
INTSQRT	2	3	1	6
EUCGCD	2	3	1	6
MFMUL	3	3	2	8
MFADD	3	2	2	7
Overall FPGA resources				1
Overall Timing				1
Total marks				40

Marks breakdown.

References

[1] S. S. Jadhav, C. Gloster, J. Naher, C. Doss, and Y. Kim, "An FPGA-based application-specific processor for implementing the exponential function," in *2020 SoutheastCon*, 2020, pp. 1–8.

[2] J. Podivinsky, M. imková, O. Cekan, and Z. Kotásek, "FPGA prototyping and accelerated verification of asips," in *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 2015, pp. 145–148.

[3] [Online]. Available: https://en.wikipedia.org/wiki/Euclidean_algorithm

[4] [Online]. Available: https://arxiv.org/pdf/2107.02762

[5] Q. A. Al-Haija, M. Al-Ja'fari, and M. Smadi, "A comparative study up to 1024 bit euclid's GCD algorithm FPGA implementation and synthesizing," in *2016 5th International Conference on Electronic Devices, Systems and Applications (ICEDSA)*, 2016, pp. 1–4.

[6] [Online]. Available: https://en.wikipedia.org/wiki/Minifloat

[7] S. Paschalakis and P. Lee, "Double precision floating-point arithmetic on FPGAs," in *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798)*, 2003, pp. 352–358.