Computer Architecture
CSCE 3301-01
Fall 2019
MS4 Report
November 19th, 2019
For
Dr. Cherif Salama

Background:

Project: femtoRV32

Author(s): Abdelhakim Badawy - abdelhakimbadawy@aucegypt.edu

Marwan Eid - marwanadel99@aucegypt.edu

Mohammed Abuelwafa - mohammedabuelwafa@aucegypt.edu

Description: Verilog modules that constructs the full pipelined datapath supporting all of RV32I 47 instructions with support to compressed instructions and integer multiplication and division.

Abstract:

This milestone is the final stage of the project. In the previous milestone we implemented the pipelined implementation using the single cycle implementation by adding the pipelined registers. Moreover, we implemented two of the bonus requirements which are supporting the RV32IC and RV32IM extensions. In the third milestone, there was an additional obstacle which was the need to implement a single ported memory for both the data memory and instruction memory which resulted in additional structural hazards and handling them. The handling was done by fetching an instruction each to clock cycles and that was implemented by generating a slower clock. For the final milestone, we added 2 bonuses which are supplying compressed instructions which are using 16 bits as an instruction, so we implemented a module to decompress the instruction to be a regular 32 bits. The second bonus was supporting the Multiplication instructions. In this milestone, FPGA testing was attempted, but due to time restrictions, the generation of bit stream accompanied with the schematic of the design was sufficient.

Technical Description:

- For Milestone 2, we used the Verilog modules we implemented in the lab for the single cycle datapath supporting only seven instructions, modified and added some aspects to support all the 47 instructions.
- For Milestone 3, we added the pipeline registers and replaced the instruction memory and the data memory with a single single-ported, byte-addressable 256 KB memory that has the instructions and the data.

- For Milestone 4, we added the required constraint files in order to implement the CPU on the FPGA. In addition to supporting the RV32IC and RV32IM extensions of the RV32 Integer set according to the following:

    1. RV32IC: in order to support the compressed instructions of the RV32I, we decided to add a Decompressor module that decodes the compressed instruction and converts it to its equivalent 32 bit instruction. The distinction between regular and compressed instructions is done through bits 0 and 1 of the opcode. The memory has a flag that distinguishes between both types, and according to this flag, PC gets incremented with the suitable increment and handling the instructions is done.

    2. RV32IM: in order to support the multiplication instructions we manipulated the Verilog constructs.

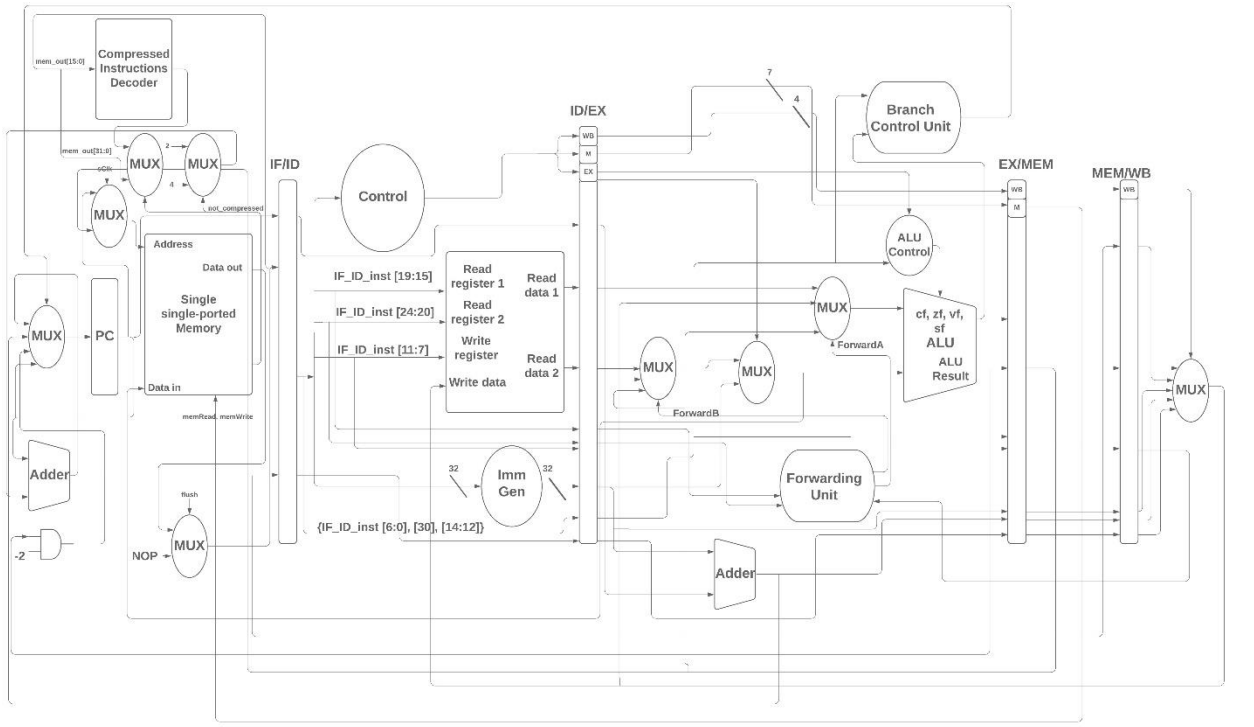
- We changed the instruction memory and the data memory to be both byte-addressable instead of word-addressable. We changed the size of the memory to be 256 KB.
- The memory is switched from dual memory system to a single ported memory architecture from which instructions and data are loaded.
- Instructions are fetched with a rate of one instruction per 2 clock cycles.
- The branch control unit is moved from the memory stage to the execution stage. This allows taking the branch decision earlier one stage and thus flushing only the instruction that is fetched after the branch if the decision is taken wrong.
- Fetching instructions every two clock cycles releases the constraints of many hazards. Flushing instructions only happens in case of a branch instruction. In case of branch, the fetched instruction is flushed in case of taking the branch.
- Memory is treated as a data memory in the second half of each clock cycle (i.e it takes the address computed from the ALU)
- We determine the specific type of the instruction in case of load or store instructions using the func3 bits of the instruction.

- We changed the size of the control signal (MemToReg) to three bits for it to be the selection line of the 8x1 multiplexer whose inputs are the ALU result, the data memory output, the immediate generator output, the output of the adder of the program counter and the immediate generator output, and the output of the adder of the program counter and the hard coded value 4. The output of this multiplexer goes to the register file write data.
- The carry flag, zero flag, overflow flag, and sign flag generated by the ALU, along with the current instruction, are inputs to the branching control unit which in turn generates a signal (pc_sel) which is an input to the control unit; this signal is an input to a module.
- The instruction needed to be implemented as a nop instruction are implemented so by setting all the control signals to 0. The NOP instruction code in hex is 00000033
- EBREAK instruction is determined by checking its opcode, func3, and the 20^{th} bit of the instruction and is implemented as a halting instruction by setting all control signals to 0 except for the branch signal which is set to 1 and is an input to a module that generates the selection line for the multiplexer whose output is the next program counter; additionally, this module has the branch signal and the instruction as inputs. The multiplexer has four inputs: the output of the adder of the program counter and the hard coded value 4, the output of the adder of the program counter and the immediate generator output, the value of the next program counter in case of a JALR instruction, and the same program counter. Accordingly, we can choose the next program counter based on the current instruction, the branch signal, and the pc_sel signal.
- The value of the next program counter in case of a JALR instruction is determined by ANDing the ALU result with -2, (32'b111…110) which thus sets the least significant bit to zero, since jumps are allowed to only even addresses.
- We implemented the two bonus features of decoding compressed instruction and adding multiplication and division instructions. The multiplication and division were made by adding their operations to the ALU and adding corresponding control signals required for the extra eight instructions.
- For the decompression, we added a module that takes compressed instructions and outputs the decompressed instruction. A multiplexer

chooses whether to take the decompressed instruction or the compressed instruction.

- For the multiplication and division instructions, they have the same opcode as R-format instructions, and some instructions have the same func3 field to those in E-format instructions. The only difference in in the 20th bit which what we use to differentiate between multiplication and division instructions and corresponding R-format instructions.

Block Diagram*:



Supported Instructions:

- Compressed Instructions:

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0	
000	0	0		0		01	C.NOP
000	nzimm[5]	rs1/rd≠0		nzimm[4:0]		01	C.ADDI *(HINT, nzimm=0)*
001	imm[11|4|9:8|10|6|7|3:1|5]					01	C.JAL *(RV32)*
001	imm[5]	rs1/rd≠0		imm[4:0]		01	C.ADDIW *(RV64/128; RES, rd=0)*
010	imm[5]	rd≠0		imm[4:0]		01	C.LI *(HINT, rd=0)*
011	nzimm[9]	2		nzimm[4|6|8:7|5]		01	C.ADDI16SP *(RES, nzimm=0)*
011	nzimm[17]	rd≠{0, 2}		nzimm[16:12]		01	C.LUI *(RES, nzimm=0; HINT, rd=0)*
100	nzuimm[5]	00	rs1′/rd′	nzuimm[4:0]		01	C.SRLI *(RV32 NSE, nzuimm[5]=1)*
100	0	00	rs1′/rd′	0		01	C.SRLI64 *(RV128; RV32/64 HINT)*
100	nzuimm[5]	01	rs1′/rd′	nzuimm[4:0]		01	C.SRAI *(RV32 NSE, nzuimm[5]=1)*
100	0	01	rs1′/rd′	0		01	C.SRAI64 *(RV128; RV32/64 HINT)*
100	imm[5]	10	rs1′/rd′	imm[4:0]		01	C.ANDI
100	0	11	rs1′/rd′	00	rs2′	01	C.SUB
100	0	11	rs1′/rd′	01	rs2′	01	C.XOR
100	0	11	rs1′/rd′	10	rs2′	01	C.OR
100	0	11	rs1′/rd′	11	rs2′	01	C.AND
100	1	11	rs1′/rd′	00	rs2′	01	C.SUBW *(RV64/128; RV32 RES)*
100	1	11	rs1′/rd′	01	rs2′	01	C.ADDW *(RV64/128; RV32 RES)*
100	1	11	—	10	—	01	*Reserved*
100	1	11	—	11	—	01	*Reserved*
101	imm[11|4|9:8|10|6|7|3:1|5]					01	C.J
110	imm[8|4:3]	rs1′		imm[7:6|2:1|5]		01	C.BEQZ
111	imm[8|4:3]	rs1′		imm[7:6|2:1|5]		01	C.BNEZ

15–13	12–11–10	9–8–7	6–5	4–3–2	1–0	
000	0			0	00	*Illegal instruction*
000	nzimm[5:4|9:6|2|3]			rd′	00	C.ADDI4SPN *(RES, nzimm=0)*
001	imm[5:3]	rs1′	imm[7:6]	rd′	00	C.FLD *(RV32/64)*
001	imm[5:4|8]	rs1′	imm[7:6]	rd′	00	C.LQ *(RV128)*
010	imm[5:3]	rs1′	imm[2|6]	rd′	00	C.LW
011	imm[5:3]	rs1′	imm[2|6]	rd′	00	C.FLW *(RV32)*
011	imm[5:3]	rs1′	imm[7:6]	rd′	00	C.LD *(RV64/128)*
100	—				00	*Reserved*
101	imm[5:3]	rs1′	imm[7:6]	rs2′	00	C.FSD *(RV32/64)*
101	imm[5:4|8]	rs1′	imm[7:6]	rs2′	00	C.SQ *(RV128)*
110	imm[5:3]	rs1′	imm[2|6]	rs2′	00	C.SW
111	imm[5:3]	rs1′	imm[2|6]	rs2′	00	C.FSW *(RV32)*
111	imm[5:3]	rs1′	imm[7:6]	rs2′	00	C.SD *(RV64/128)*

15–13	12	11–10–9–8–7	6–5–4–3–2	1–0	
000	nzuimm[5]	rs1/rd≠0	nzuimm[4:0]	10	C.SLLI *(HINT, rd=0; RV32 NSE, nzuimm[5]=1)*
000	0	rs1/rd≠0	0	10	C.SLLI64 *(RV128; RV32/64 HINT; HINT, rd=0)*
001	uimm[5]	rd	uimm[4:3|8:6]	10	C.FLDSP *(RV32/64)*
001	uimm[5]	rd≠0	uimm[4|9:6]	10	C.LQSP *(RV128; RES, rd=0)*
010	uimm[5]	rd≠0	uimm[4:2|7:6]	10	C.LWSP *(RES, rd=0)*
011	uimm[5]	rd	uimm[4:2|7:6]	10	C.FLWSP *(RV32)*
011	uimm[5]	rd≠0	uimm[4:3|8:6]	10	C.LDSP *(RV64/128; RES, rd=0)*
100	0	rs1≠0	0	10	C.JR *(RES, rs1=0)*
100	0	rd≠0	rs2≠0	10	C.MV *(HINT, rd=0)*
100	1	0	0	10	C.EBREAK
100	1	rs1≠0	0	10	C.JALR
100	1	rs1/rd≠0	rs2≠0	10	C.ADD *(HINT, rd=0)*
101	uimm[5:3|8:6]		rs2	10	C.FSDSP *(RV32/64)*
101	uimm[5:4|9:6]		rs2	10	C.SQSP *(RV128)*
110	uimm[5:2|7:6]		rs2	10	C.SWSP
111	uimm[5:2|7:6]		rs2	10	C.FSWSP *(RV32)*
111	uimm[5:3|8:6]		rs2	10	C.SDSP *(RV64/128)*

- Regular instructions:

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

- Multiplication and Division Instructions:

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Simulation screenshot:

Notes:

- The schematic is attached in the submission folder.
- A PDF file for the block diagram is attached in the report folder. Here is a shared link for the block diagram for better tracing of the wires.
- Here is a sharable link for the block diagram for better tracing the wires: https://www.lucidchart.com/invitations/accept/6c98dee8-1bb2-414f-9e61-f0ad342917a6
- Test codes for the supported instructions, including the compressed instructions and the multiplication and division instructions, are attached in the test folder.
- A C++ program that generates hex files to be loaded into the memory from the assembly code is attached.
- Waveforms for the simulation are attached.

Conclusion:

We were able to implement the pipelined processor supporting the 47 instructions, compressed instructions, and multiplication and division instructions. Simulation testing and design implementation were successfully completed; bitstream for the project was successfully generated as well; however, testing on the FPGA did not work as expected.