Bangladesh University of Engineering & Technology

Design and Simulation of a 32-bit RISC-V Core and APB

Course No: EEE 468

Course Title: VLSI Laboratory

Group No: 03

Group Members:

Md. Rafiqul Islam Rafi	- 1706007
Sheikh Munim Hussain	- 1706008
Abdullah Al Mahmood	- 1706010
Md. Jahidul Hoq Emon	- 1706017
Shafin Shadman Ahmed	- 1706020
Golam Mahmud Samdani	- 1706029
Ayan Biswas Pranta	- 1706032

Submitted To: Dr. A.B.M Harun-Ur-Rashid, Mumtahina Islam Sukanya

Introduction:

VLSI, or Very Large Scale Integration, involves the process of incorporating transistors onto a single semiconductor chip, with programming codes being the most effective method for achieving this. SystemVerilog is a crucial coding language that enables the simulation of numerous transistors on a single chip through coding.

There are different types of microprocessor architectures such as single-cycle, multi-cycle, and pipelined. The multi-cycle architecture specifies its input and output Datapath and controls, and SystemVerilog can be utilized for designing this type of architecture.

RISC-V is an open standard instruction set architecture that follows the principles of RISC. It is free to use and can be easily implemented with SystemVerilog. RISC-V has its own method of defining instructions in assembly and machine languages, and codes are compiled in SystemVerilog based on RISC-V standards.

For this particular project, the instructions were compiled in RISC-V architecture using SystemVerilog.

The architecture here we used is Von Neumann architecture.

Interface: APB

APB or Advanced Peripheral Bus is a industry standard protocol that is used to communicate with peripheral components.

- APB Bridge is the master
- Peripherals are the slave
- Write means master to slave communication
- Read means slave to master communication
- Separately passing address and data, much faster

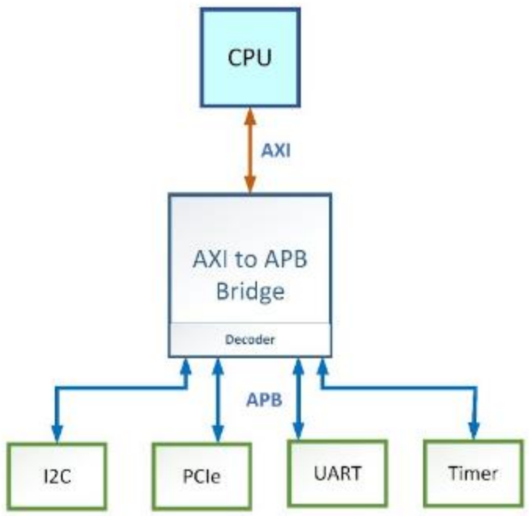


Figure 1: APB Protocol Mounted to CPU

The finite state machine or fsm diagram of APB protocol is shown below:

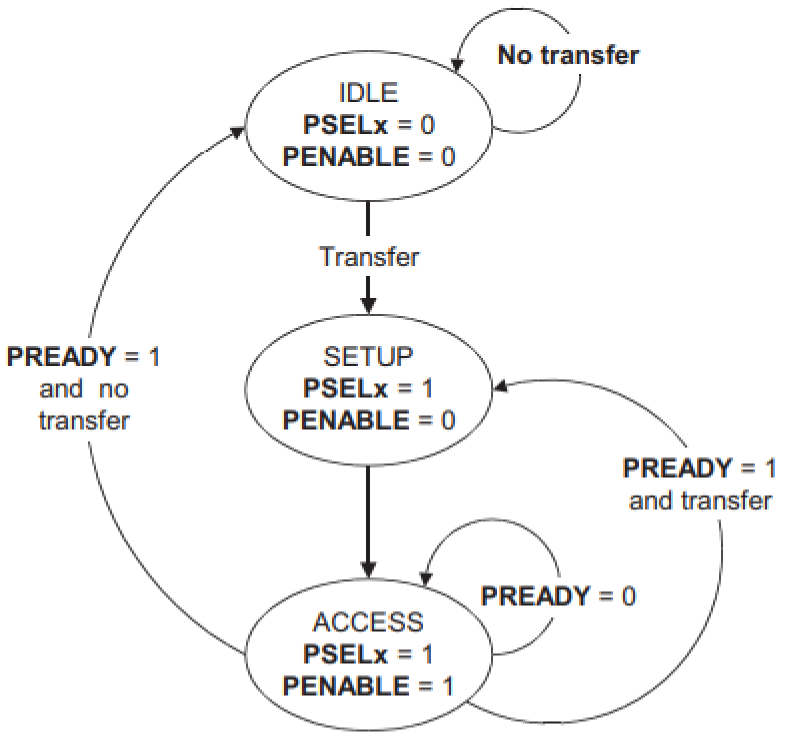


Figure 2: Working fsm of APB

The main working principle of APB is laid on transfer signal. When no transfer and in reset, the APB works in IDLE state. Then it goes to SETUP state when it gets a transfer signal. Then it goes to ACCESS state in which the data is accessed. Then depending on PREADY and transfer signal, the state changes to the 3 states.

Now we will look into the block diagram of memory wrapper.

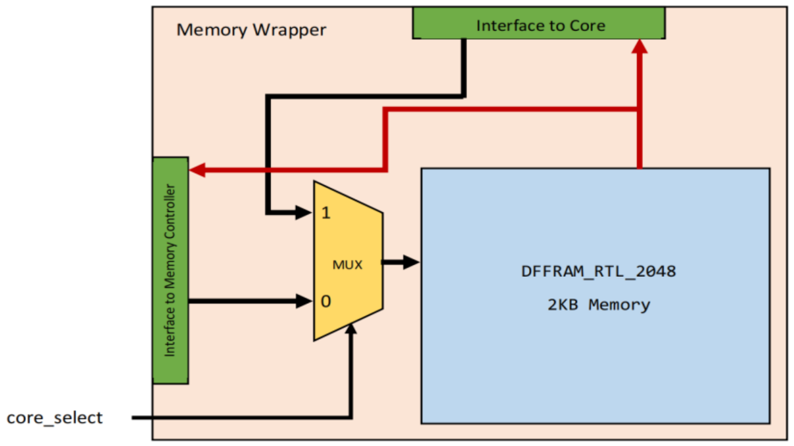


Figure 3: Memory Wrapper Block Diagram

The memory wrapper connects the memory in two ways. In one way, the memory is connected to interface to memory controller which takes input from outer sources such as instructions. In the other way, memory takes or sends data from or to core. Here, we have used a 2kB memory.

Core

Now we will deeply look into core parts by parts. At first we will see the arithmetic and logic unit part.

ALU:

Arithmetic operations like addition, subtraction and operations are done by ALU block. The block diagram of ALU is given below:

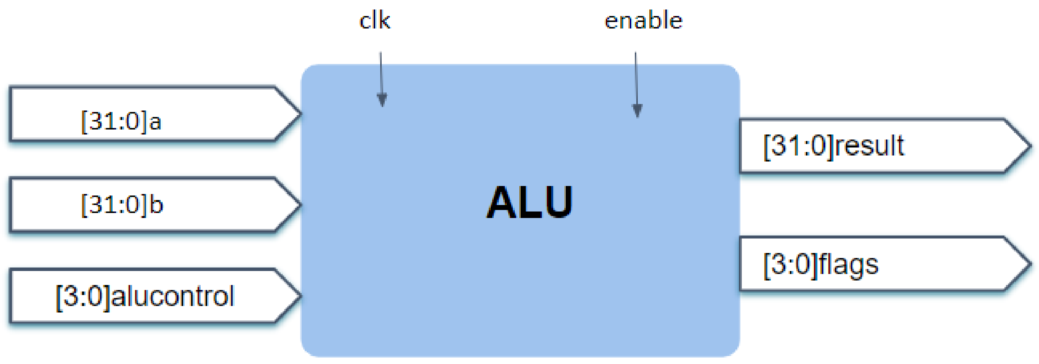


Figure 4: ALU Block diagram

Along with result outputs, ALU also give flags that are used to know whether the core will take a conditional branch statement or not. The description of flags is shown below:

Flags	**Description**
N	Set when the result of operation is negative
Z	Set when the result of operation is zero
C	Set when carry is generated
V	set when overflow occurs

Which operation of arithmetic and logic unit will be done is set by ALU Decoder unit. The operation of ALU decoder unit is summarized in the table given below:

ALUOp	funct3	Op_5*func7_5	func7_b_5	ALUControl	Instruction
00	x	x	x	4’b0000	add
01	x	x	x	4’b0001	subtract
10	3’b000	1	x	4’b0001	subtract
		0	x	4’b0000	add
	3’b001	x	x	4’b0110	sll.slli
	3’b010	x	x	4’b0101	slt,slti
	3’b100	x	x	4’b0100	xor,xori
	3’b101	x	1	4’b1000	sra,srai
		x	0	4’b0111	srl,srli
	3’b110	x	x	4’b0011	or,ori
	3’b111	x	x	4’b0010	and,andi

The ALU Decoder block can be viewed as the following one:

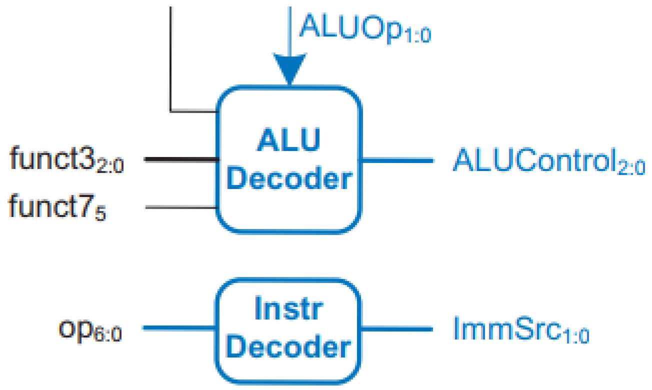


Figure 5: ALU Decoder Working Block Diagram

Non-Architectural Element:

Now, we will look at the non-architectural element of our core. There are two types of flip flops used in the core. One is without enable signal and another is with enable signal.

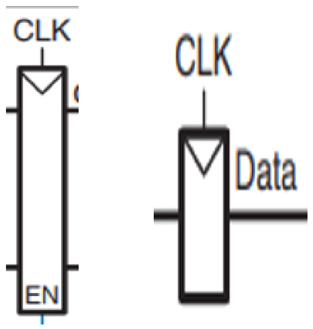


Figure 6: Flip Flop with enable signal (left) and without enable signal (right)

Both of the flip flops have the reset signal that are not shown in the diagram.

Register File:

Then we will look into register file used in the core.

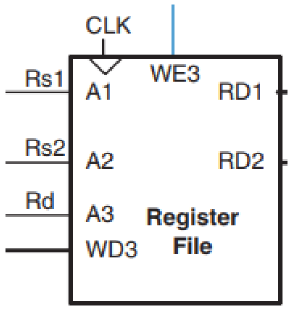
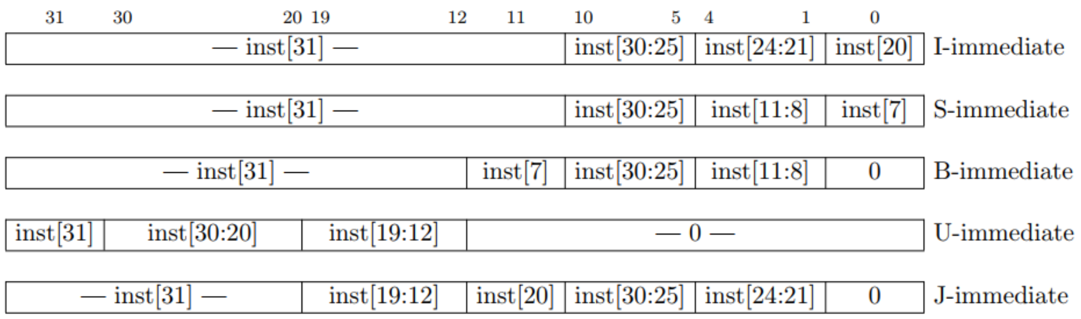


Figure 7: Register File

The register file used here, takes instruction's 20th to 16th position as Rs1, 25th to 21st position as Rs2 and 12th to 8th position as Rd.

Immediate Extender:

In immediate extender, immediate is extended depending on which operation to do using the immediate.



31	30 … 20	19 … 12	11	10 … 5	4 … 1	0	
— inst[31] —	— inst[31] —	— inst[31] —	— inst[31] —	inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —	— inst[31] —	— inst[31] —	— inst[31] —	inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —	— inst[31] —	— inst[31] —	inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —	— 0 —	— 0 —	— 0 —	U-immediate
— inst[31] —	— inst[31] —	inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

Figure 8: Immediate extension mapping for different types of operation

Data Path:

The overall data path of our system is shown below:

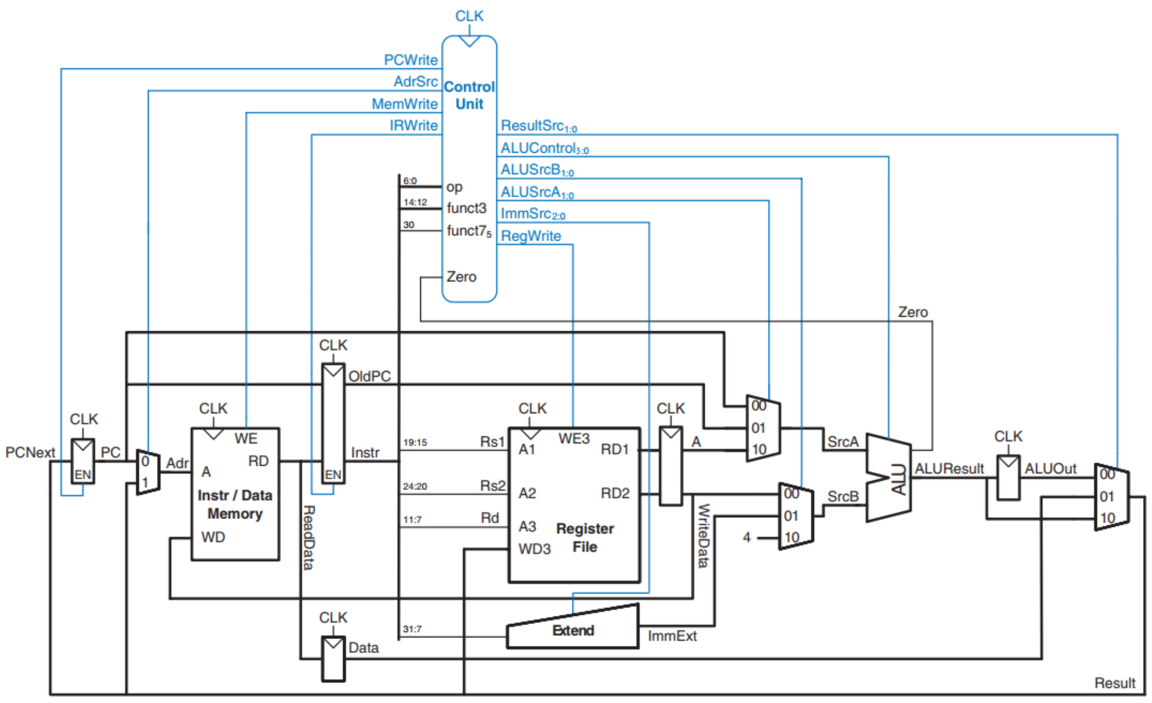


Figure 9: Datapath and core assembly of our system

Branch Unit:

The branch unit decides whether to take a branch instruction or not depending the flags got from ALU. This unit just gives the output flag which determines whether we need to branch or not from a branch instruction.

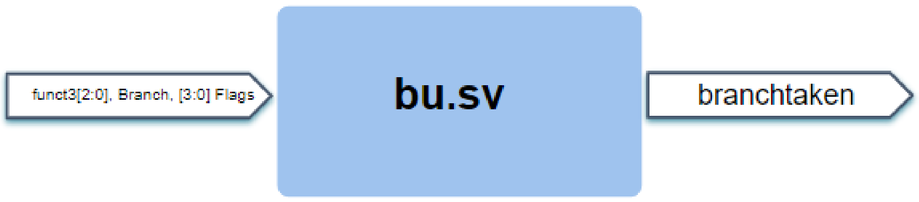


Figure 10: Branch unit diagram

The branch unit analyzes the values of flags n, z, c and v from the ALU operation and depending on the following table expression it decides whether we need to branch or not.

funct3	condition	operation
3’b000	z	beq
3’b001	~z	bne
3’b100	n^v	blt
3’b101	~(n^v)	bge
3’b110	~c	bltu
3’b111	c	bgeu

Load/Store operation module:

Whether an instruction will load a value from memory or will it store a value in memory is decided by the logics given below:

Instruction	funct3	LoadType/StoreType
Store	000	01
Neither	010	00
Load	100	1x

The branching unit, ALU decoder, instruction decoder, load store decoder and main finite state machine of the core makes the overall core control system. Now we will look at the fsm of our core.

Main FSM for control:

For main core FSM signal, we need to start any instruction at FETCH state. Whenever our core is reset or if the state value is beyond our total state number (11), then our core will automatically start from FETCH state. After FETCH we will go to the decode state. At the starting of DECODE state, the program counter is updated. From DECODE state, the type of instruction is decoded and as a result, the fsm goes to various other states depending on the instruction type decoded. For example, if the core gets an addi (or adding with immediate signal) then it will follow the states: FETCH > DECODE > EXECUTEI > ALUWB. After the ALUWB or ALU writeback state, the value is written in the corresponding register and the instruction ends. The fsm is shown below:

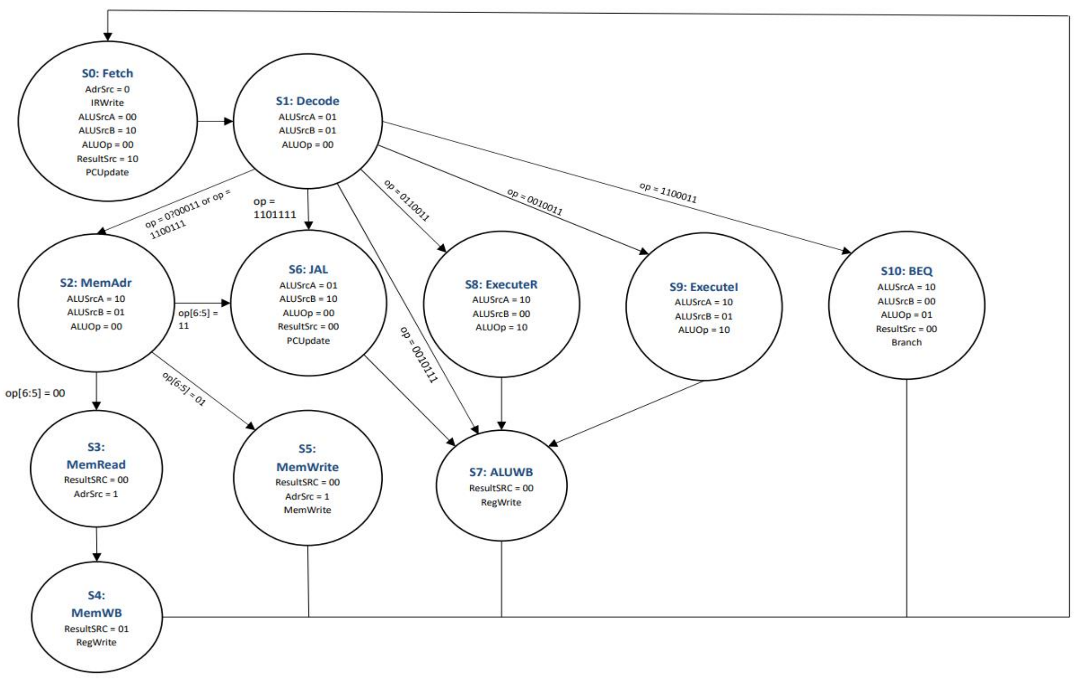
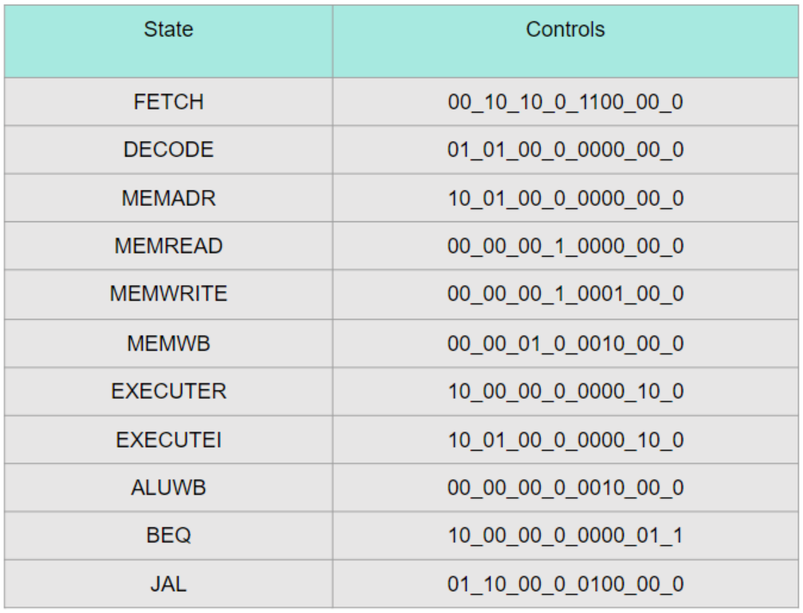


Figure 11: FSM of the core

We can see that, for lw (load) instruction, we need 5 states or 5 full cycles. For JALR instruction, we also need 5 states. For branching instruction, we need 3 states. Without these, we need 4 states for most other instructions. The control signals for these states are shown below:



State	Controls
FETCH	00_10_10_0_1100_00_0
DECODE	01_01_00_0_0000_00_0
MEMADR	10_01_00_0_0000_00_0
MEMREAD	00_00_00_1_0000_00_0
MEMWRITE	00_00_00_1_0001_00_0
MEMWB	00_00_01_0_0010_00_0
EXECUTER	10_00_00_0_0000_10_0
EXECUTEI	10_01_00_0_0000_10_0
ALUWB	00_00_00_0_0010_00_0
BEQ	10_00_00_0_0000_01_1
JAL	01_10_00_0_0100_00_0

The control signal values consist of ALUSrcA (2 bits), ALUSrcB (2 bits), ResultSrc (2 bits), AdrSrc, IRWrite, PCUpdate, Regwrite, MemWrite, ALUOp (2 bits), Branch signals. The signals that are not mentioned are of single bit.

Immediate Extender and Instruction Decoder:

For decoding instructions how should be extended, we need and instruction decoder. This block works with the principle shown in the box below:

Instruction	Op Code	ImmSrc
R-type	0110011	xxx
I-Type ALU	0010011	000
lw	0000011	000
sw	0100011	001
be (branches)	1100011	010
jal	1101111	011
jalr	1100111	000
auipc	001011	100

Taking the ImmSrc signal, immediate is extended which is based on which type of operation is done. The detailed immediate extension algorithm is shown below:

ImmSrc	ImmExt	Type	Description
000	{{20{instr[31]}}, instr[31:20]}	I	12-bit signed immediate
001	{{20{instr[31]}}, instr[31:25], instr[11:7]}	S	12-bit signed immediate
010	{{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0}	B	13-bit signed immediate
011	{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}	J	21-bit signed immediate
100	{instr[31:12], 12'b0}	U	20-bit upper immediate to PC

Sometimes zero is needed to be extended. In this case 0 is extended up to first 24 bits from MSB. For this we used zero extend block.

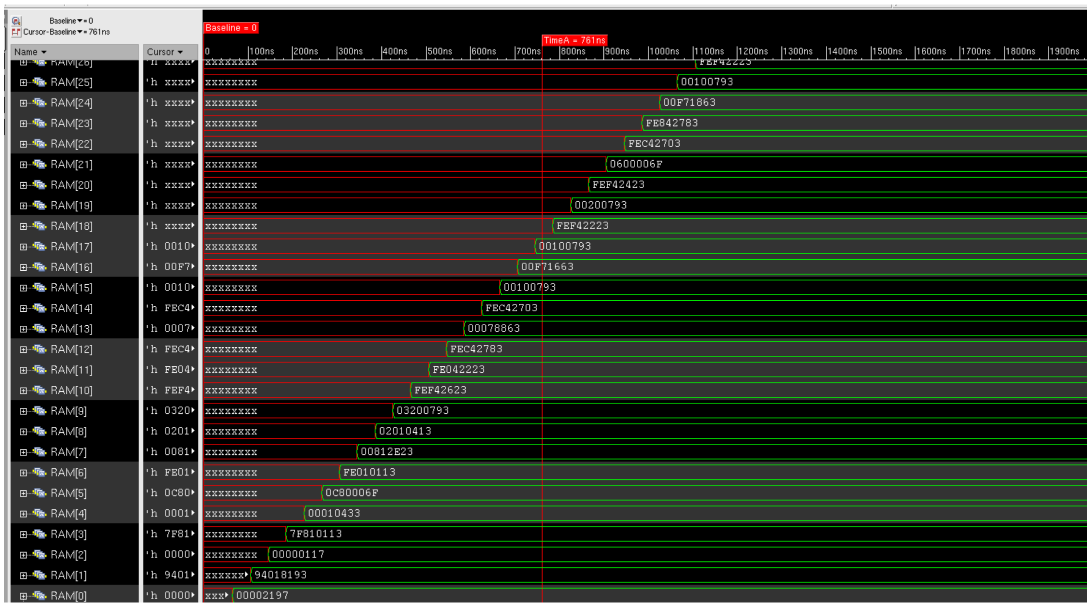
Then we accumulate the controller and the Datapath to create our overall processor. In this process, we need to instantiate the controller and Datapath code in code of our main processor.

Next, we will connect the created processor with the memory module and memory wrapper. In this case, an input signal calling ‘core_select’ will be used to determine whether we will use the core or stop the processor to read and store data in the memory.

At a top testbench module will be used to see whether our overall system works or not. In this part we will have reset_event task which will reset the hole system at first for fresh start. Then we access the instructions and get the number of lines the instructions are written. Then we write the instructions in the memory/ram using APB. At last, we run the core after resetting it again and watch the outputs.

Results

In this part the result of testbench simulation is given. At first we will look at the data uploading in RAM.



Baseline = 0
Cursor-Baseline = 761ns
Name
Cursor
TimeA = 761ns
RAM[26] 'h xxxx xxxxxxxx FEF42223
RAM[25] 'h xxxx xxxxxxxx 00100793
RAM[24] 'h xxxx xxxxxxxx 00F71863
RAM[23] 'h xxxx xxxxxxxx FE842783
RAM[22] 'h xxxx xxxxxxxx FEC42703
RAM[21] 'h xxxx xxxxxxxx 0600006F
RAM[20] 'h xxxx xxxxxxxx FEF42423
RAM[19] 'h xxxx xxxxxxxx 00200793
RAM[18] 'h xxxx xxxxxxxx FEF42223
RAM[17] 'h 0010 xxxxxxxx 00100793
RAM[16] 'h 00F7 xxxxxxxx 00F71663
RAM[15] 'h 0010 xxxxxxxx 00100793
RAM[14] 'h FEC4 xxxxxxxx FEC42703
RAM[13] 'h 0007 xxxxxxxx 00078863
RAM[12] 'h FEC4 xxxxxxxx FEC42783
RAM[11] 'h FE04 xxxxxxxx FE042223
RAM[10] 'h FEF4 xxxxxxxx FEF42623
RAM[9] 'h 0320 xxxxxxxx 03200793
RAM[8] 'h 0201 xxxxxxxx 02010413
RAM[7] 'h 0081 xxxxxxxx 00812E23
RAM[6] 'h FE01 xxxxxxxx FE010113
RAM[5] 'h 0C80 xxxxxxxx 0C80006F
RAM[4] 'h 0001 xxxxxxxx 00010433
RAM[3] 'h 7F81 xxxxxxxx 7F810113
RAM[2] 'h 0000 xxxxxxxx 00000117
RAM[1] 'h 9401 xxxxxx 94018193
RAM[0] 'h 0000 xxx 00002197

Figure 12: Uploading instructions in RAM/Memory

We can see that the instructions are uploaded in the memory one by one sequentially and all of the bits from one instructions at a time. It is the advantage of APB protocol.

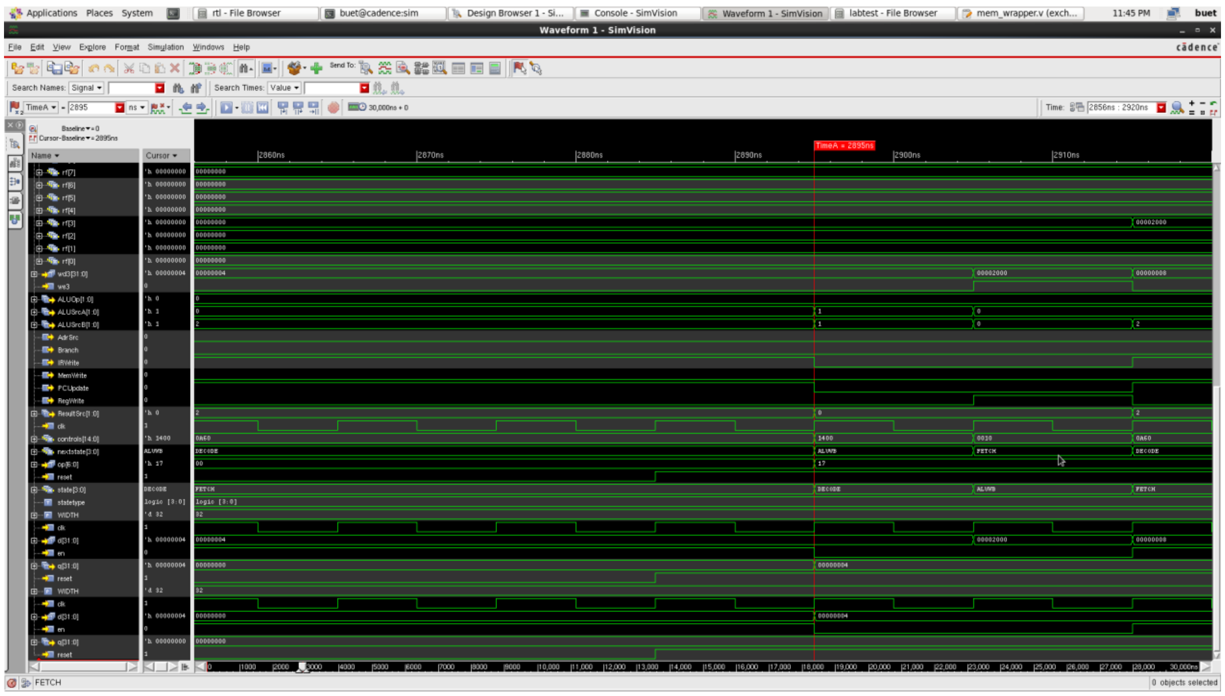


Figure 13: Instructions uploading is done

Here, in the above image, we can see that after 2895ns, the uploading of the instruction is done. Just one cycle before, the reset signal is made 0 to 1 so that the core is reset and starts from a fresh state. We also can see that at 2895ns, the state goes from FETCH to decode and after ALUWB state, the output is updated.

Now we will see that the value of r[12] becomes 1 as the input number is not a prime number (50 in this case).

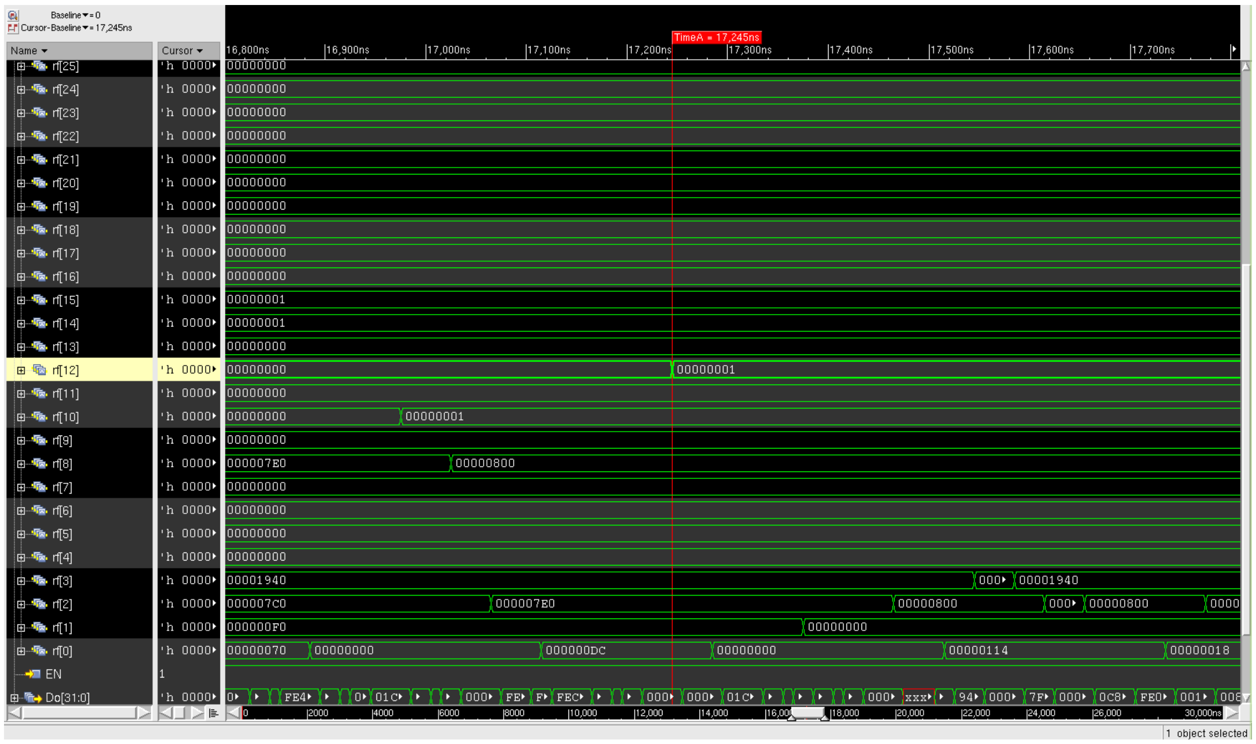


Figure 14: r[12] becomes 1

We can see that, we need **17.245 µs** to reach that point where r[12] becomes 1.

Now we will see the description of different instructions used in the testbench code. At first we will look at the auipc statement.

Statement: 0: auipc gp, 0x2

Here, gp means r[4]. At first pc is 0. Adding 2 to the upper 20 bits of program counter gives a value equal to 0x00002000.

If we look at the output timing diagram, we can see that, after ALUWB this value is written on r[4].

Here, the auipc instruction takes 3 cycles which is determined by the 3 stages shown in fsm earlier.

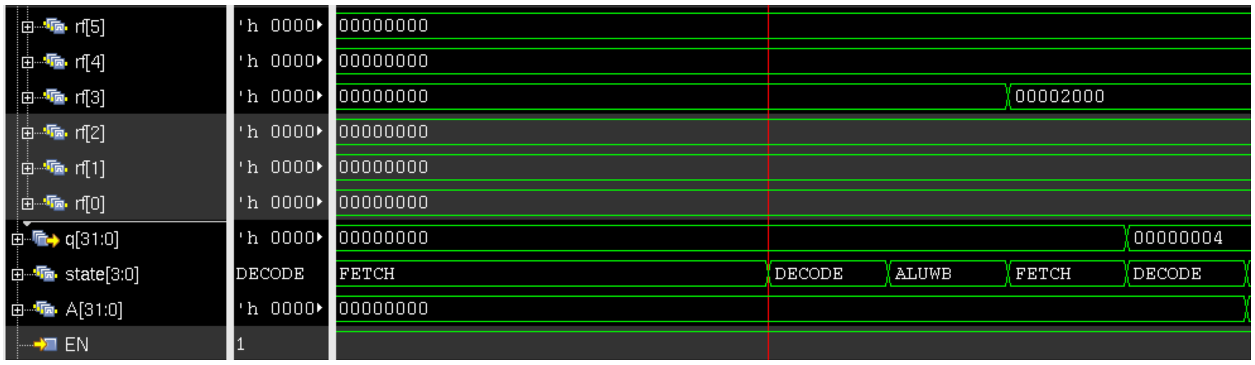


Figure 15: auipc instruction

Statement: 10: add s0, sp, zero

Here, the values at r[0] and at r[2] is added and stored in r[8]. Correspondingly output should be 800 and it is verified by the timing diagram. This time also output is updated after ALUWB state. This instruction lasts for 4 different cycles.

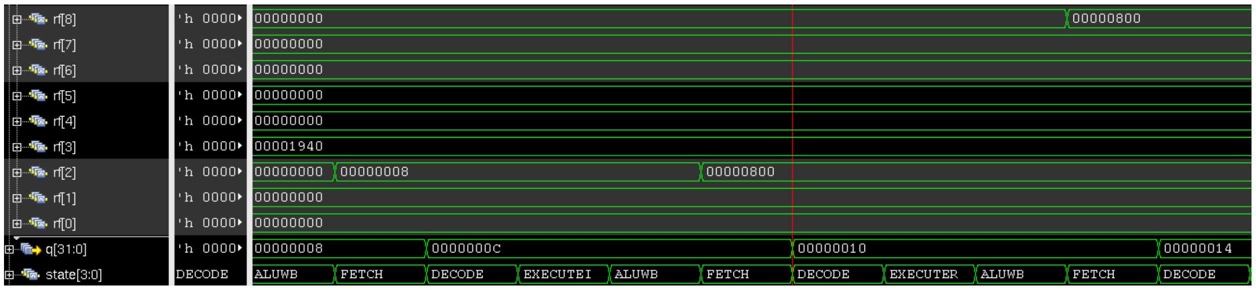


Figure 16: add statement

Statement: 14: j dc <main>

This is the code for unconditional jump. Here the code jumps to PC = 0xdc after executing this instruction. The next program counter after 14 is dc for this case.

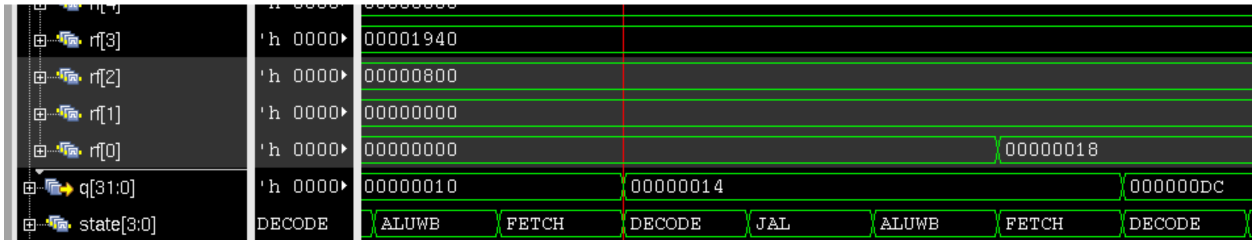


Figure 17: Unconditional Jump Operation

Statement: 1c: sw s0, 28(sp)

In this statement, store operation is done at the point of the ram pointing by 28 + value of r[2] and right shifted by 2.

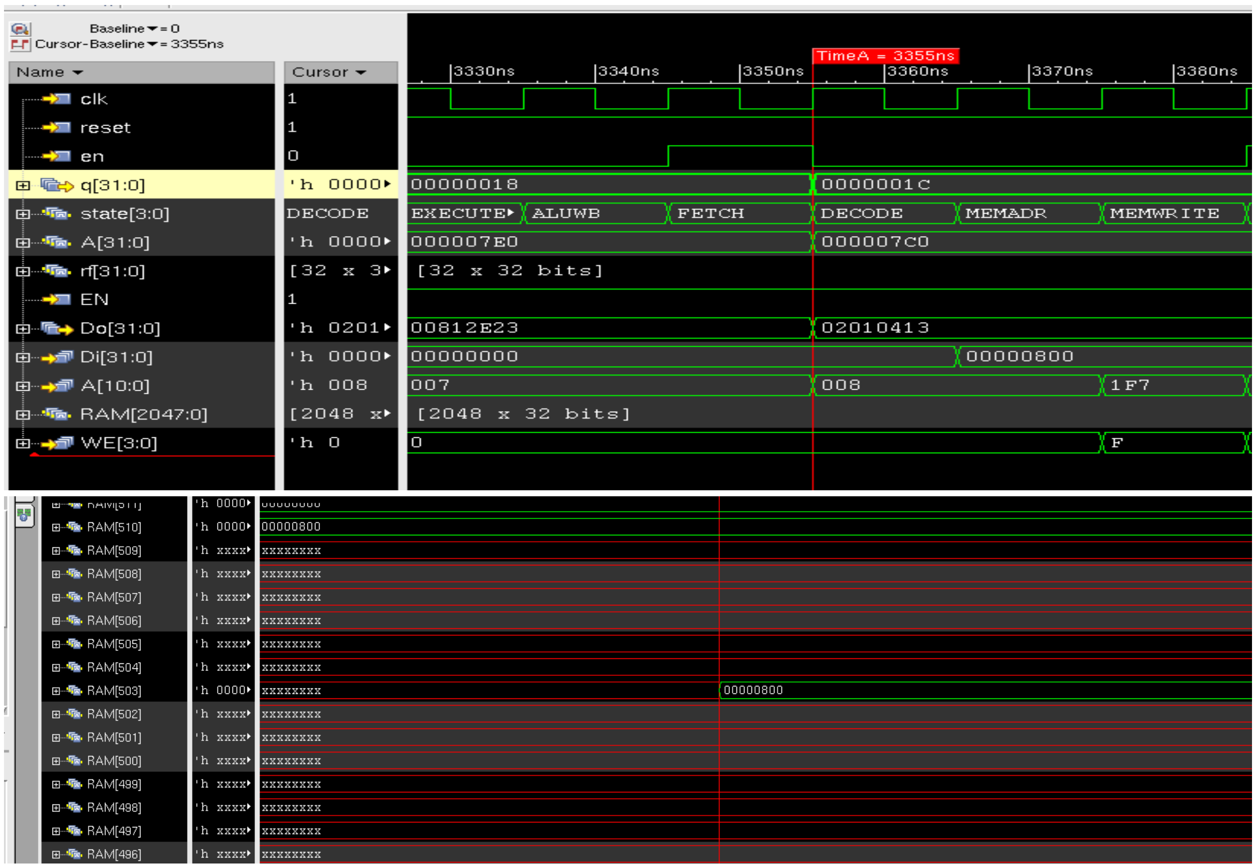


Figure 18: 800 is stored in the preferred ram location

Statement: 20: addi s0, sp, 32

This is originally the add instruction whose second operand is an immediate. All the other signals are similar to add instruction given couple of pages above.

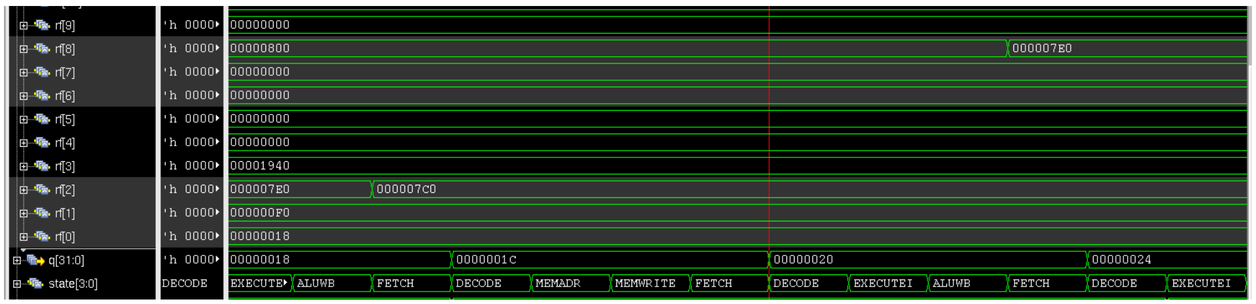


Figure 19: addi instruction

Statement: 24: li a5, 50

This is originally a pseudo instruction of addi operation. Here, the value of the immediate (here 50) is stored in r[15]. In the picture we can see the value as 32 because it is shown in hexadecimal and hexadecimal value of binary 50 is 32. The value is written after ALUWB operation.

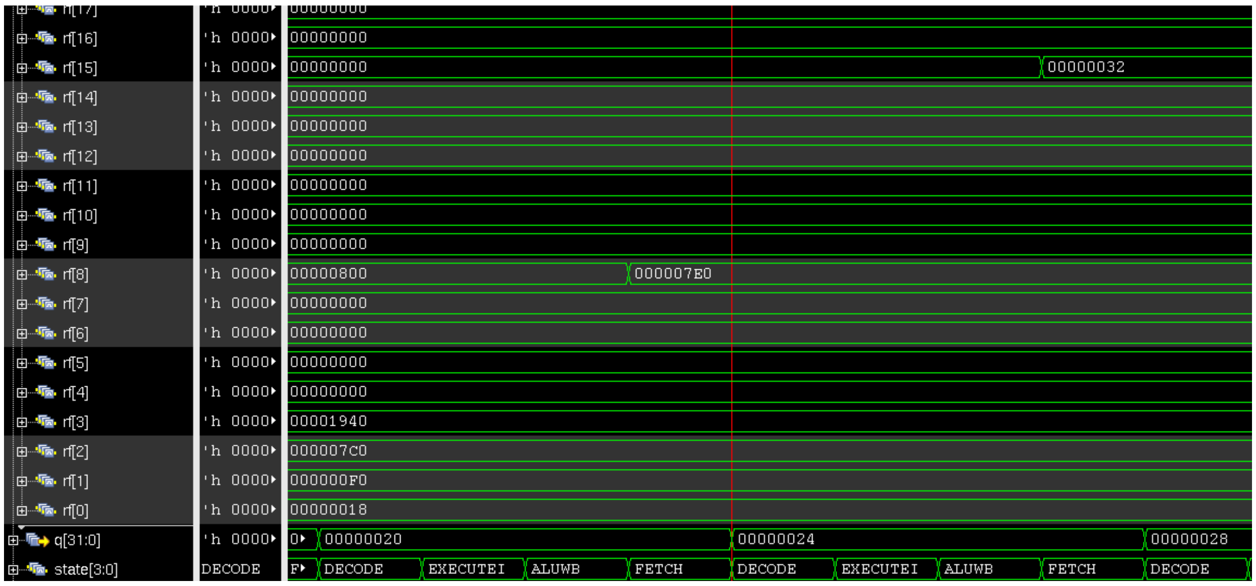


Figure 20: li operation

Now, we will see two types of branching instruction, one is branching happening and other one is the opposite event.

Statement: 34: beqz a5,44 <prime_number+0x2c>
As value at r[15] is not 44, so it equality is not proved and as a result branch is not taken. Next PC is 38 for not taking the branching.

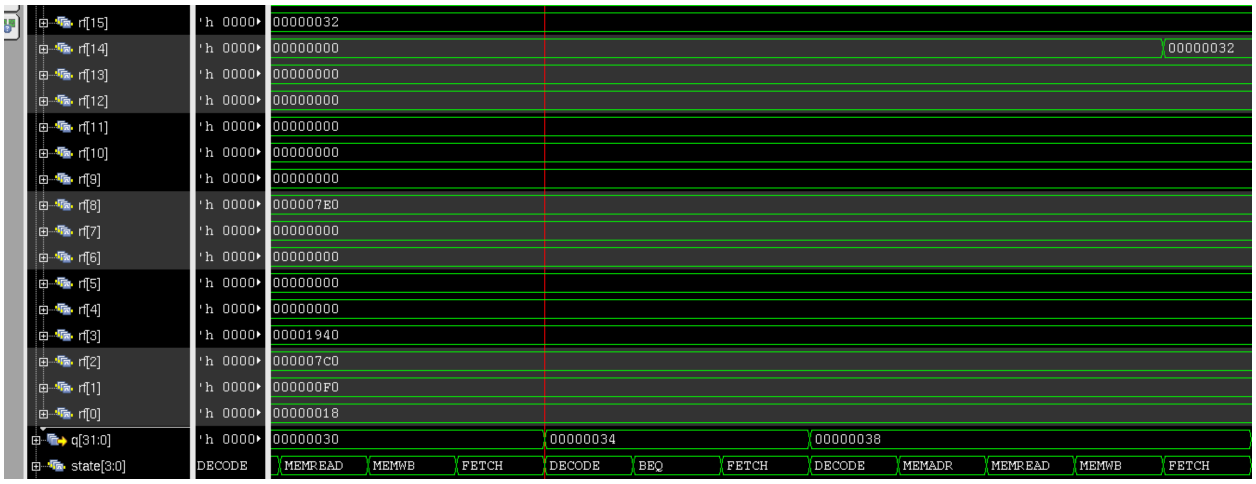


Figure 21: Branching not taken

Statement: 60: bne a4,a5,70 <prime_number+0x58>
Here, the nonequality of r[14] with the summation of r[15] and 70 is true. As a result, the branching takes place and after program counter 60, here comes 70 instead of 64,

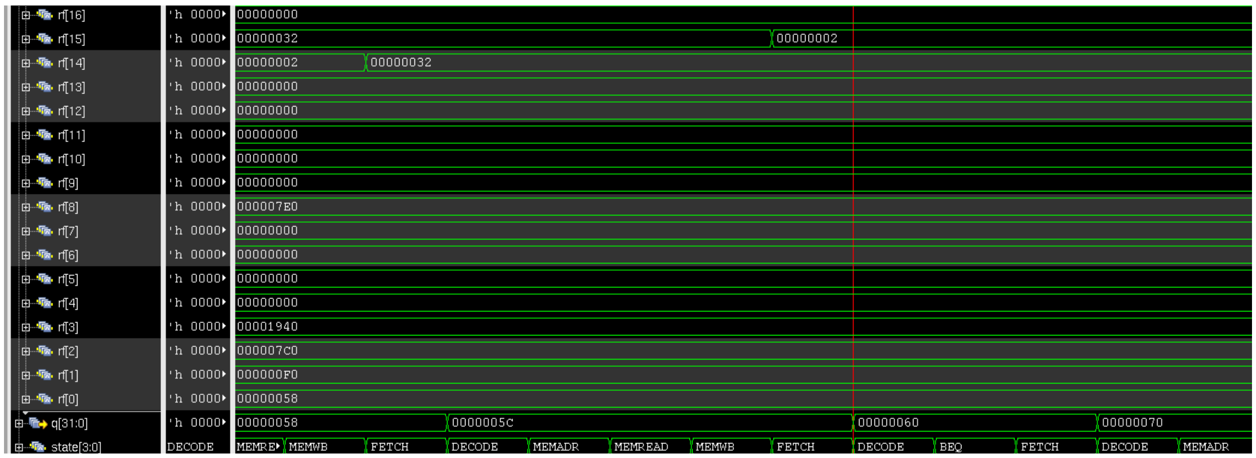


Figure 22: Branching is taken

Statement: 38: lw a4,-20(s0)

Here for load statement, value at r[8] is subtracted by 20 and right shifted by 2. Then we get a decimal digit of 499. The value at ram position 499 is 32. As a result at r[14] is set equal to 32 after MEMWB state.

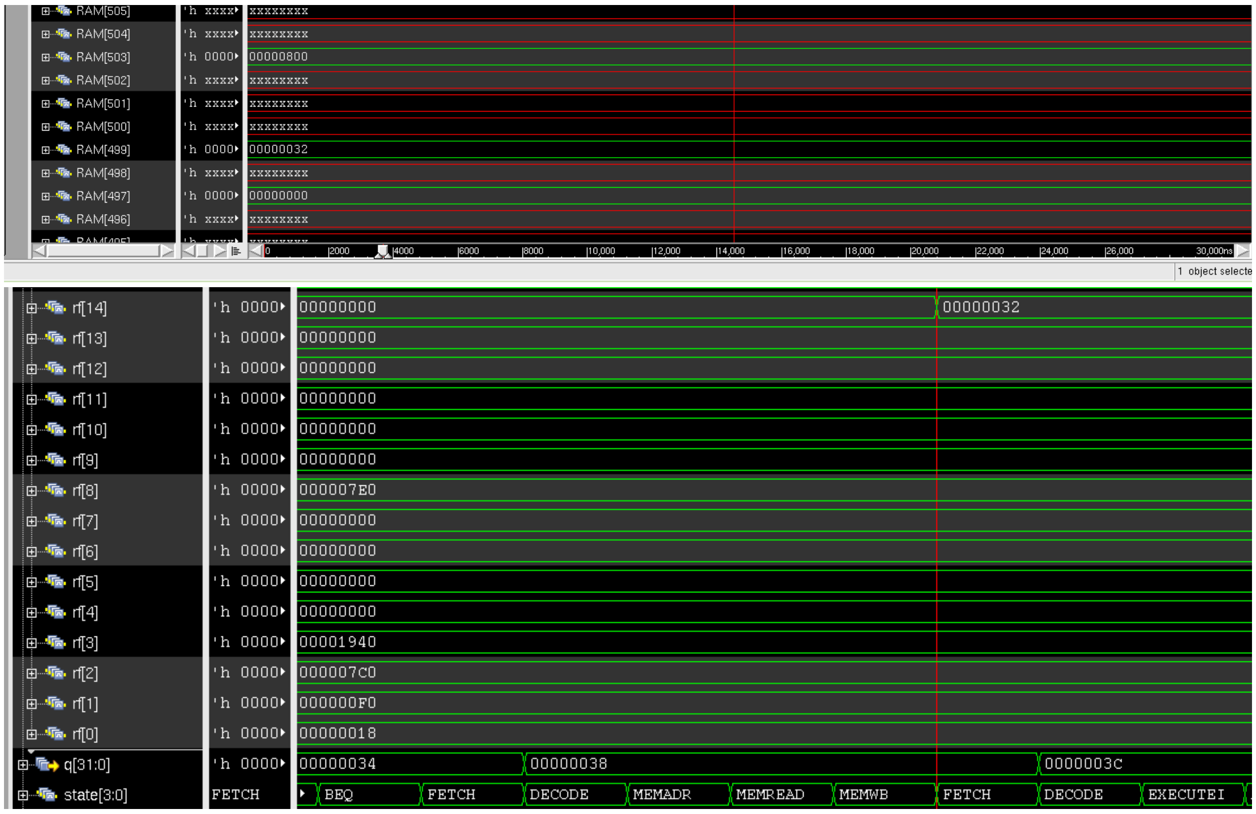

RAM[505] 'h xxxx xxxxxxxx
RAM[504] 'h xxxx xxxxxxxx
RAM[503] 'h 0000 00000800
RAM[502] 'h xxxx xxxxxxxx
RAM[501] 'h xxxx xxxxxxxx
RAM[500] 'h xxxx xxxxxxxx
RAM[499] 'h 0000 00000032
RAM[498] 'h xxxx xxxxxxxx
RAM[497] 'h 0000 00000000
RAM[496] 'h xxxx xxxxxxxx
0 2000 4000 6000 8000 10,000 12,000 14,000 16,000 18,000 20,000 22,000 24,000 26,000 30,000ns
1 object selecte
rf[14] 'h 0000 00000000 00000032
rf[13] 'h 0000 00000000
rf[12] 'h 0000 00000000
rf[11] 'h 0000 00000000
rf[10] 'h 0000 00000000
rf[9] 'h 0000 00000000
rf[8] 'h 0000 000007E0
rf[7] 'h 0000 00000000
rf[6] 'h 0000 00000000
rf[5] 'h 0000 00000000
rf[4] 'h 0000 00000000
rf[3] 'h 0000 00001940
rf[2] 'h 0000 000007C0
rf[1] 'h 0000 000000F0
rf[0] 'h 0000 00000018
q[31:0] 'h 0000 00000034 00000038 0000003C
state[3:0] FETCH BEQ FETCH DECODE MEMADR MEMREAD MEMWB FETCH DECODE EXECUTEI

Figure 23: lw operation

Statement: 90: sub a5,a4,a5

This is the same as add instruction but it shows that the result is ok when the r[14] value is subtracted by r[15] and uploaded in r[15] again. The timing diagram supports the statement.

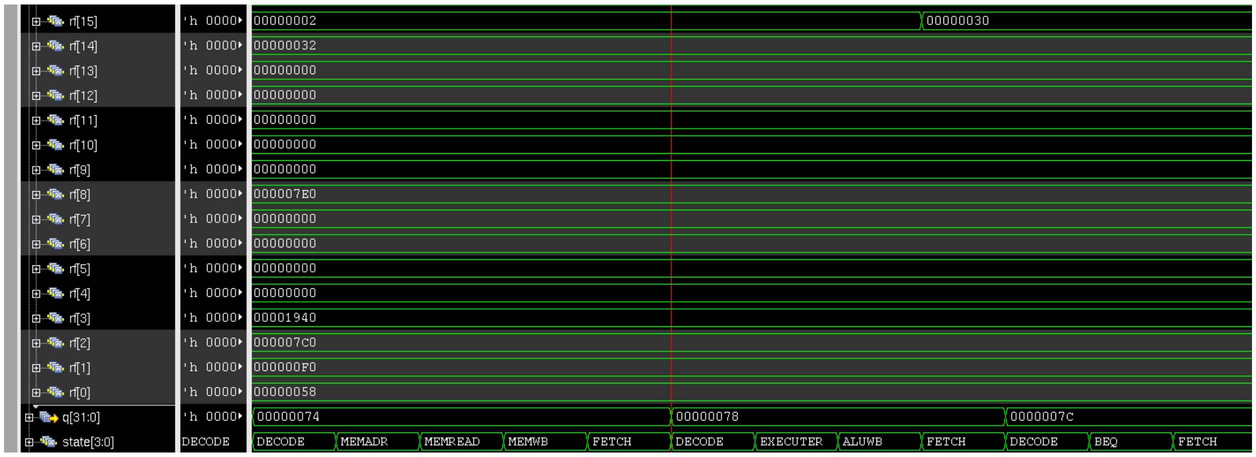


Figure 24: Subtraction operation

The last operation from testbench code is: f8: mv a2,a5

It is also a pseudo instruction of addi. This instruction gives the output r[12] = 1 that was desired first,

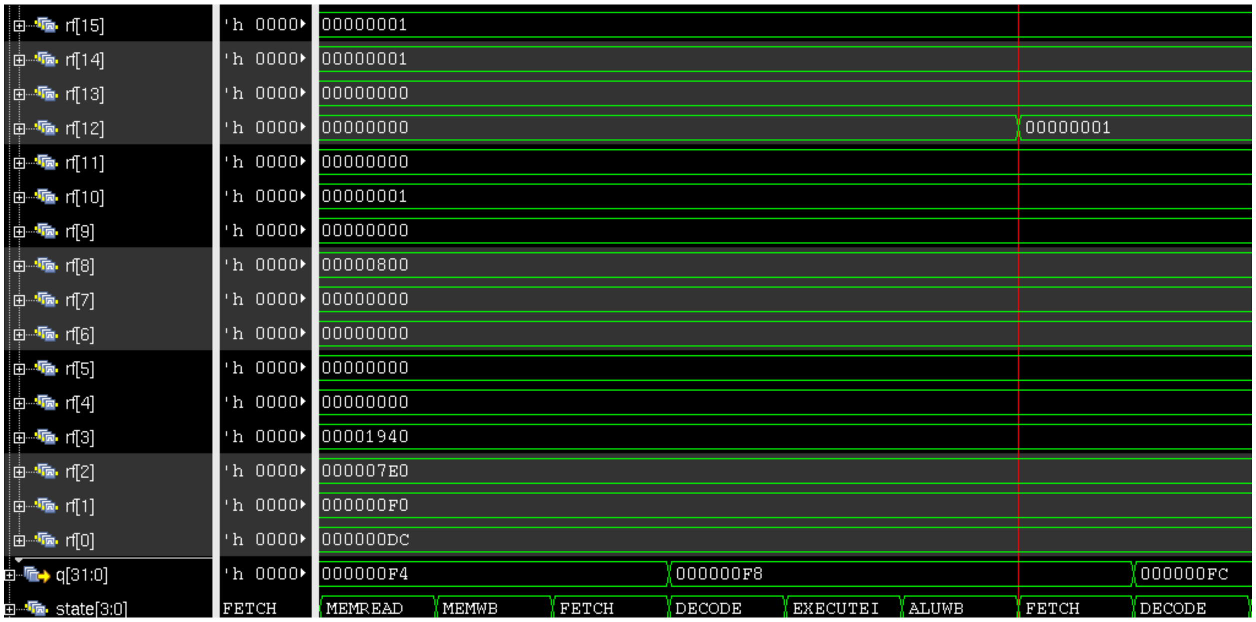


Figure 25: mv operation

So, we can see that mv operation successfully works. So, this includes all the necessary instructions required for running the testbench instruction given in the project proposal.

Next, we will add a small code snippet to see more and observe how our core works when it gets a different instruction than the proposed ones.

Example of running a new code

In this stage, we will run a small new code to see its effect. At first the code is explained below:

```
addi x2 x0 5
addi x3 x0 5
sw x3 20(x2)
nop
or x5 x2 x3
andi x6 x5 5
```

At first, after we load instructions in the memory, the core is reset and all the register values become 0. So, after first and second instruction, r[2] and r[3] should become 5.

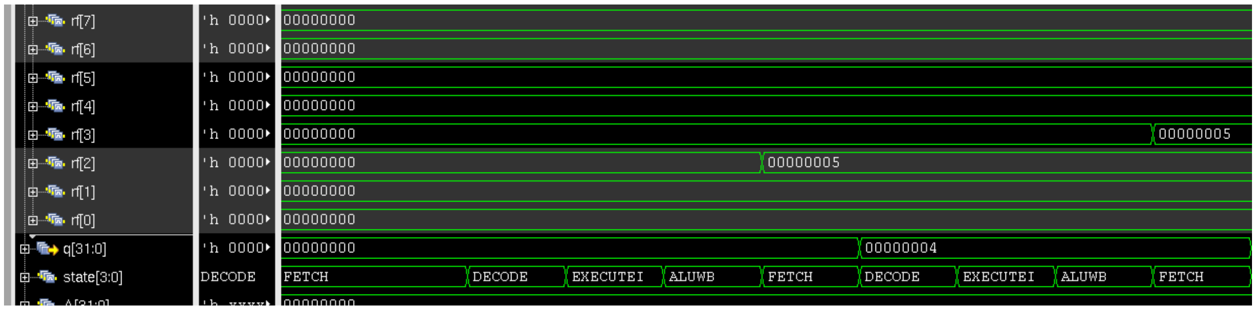

rf[7] 'h 0000 00000000
rf[6] 'h 0000 00000000
rf[5] 'h 0000 00000000
rf[4] 'h 0000 00000000
rf[3] 'h 0000 00000000 00000005
rf[2] 'h 0000 00000000 00000005
rf[1] 'h 0000 00000000
rf[0] 'h 0000 00000000
q[31:0] 'h 0000 00000000 00000004
state[3:0] DECODE FETCH DECODE EXECUTEI ALUWB FETCH DECODE EXECUTEI ALUWB FETCH

Figure 26: First two addi operations

The timing diagram given above approves that our core is working better after executing one after another. It ensures that our core can start working correctly even for different instructions.

The next instruction is storing instruction. It is similar to the instruction we showed before. So, we can skip it here.

Then there comes nop operation which doesn't do anything at all. After that we will get an OR instruction followed by an andi instruction. Both of these instructions work in similar way, so we will show the or operation here only.

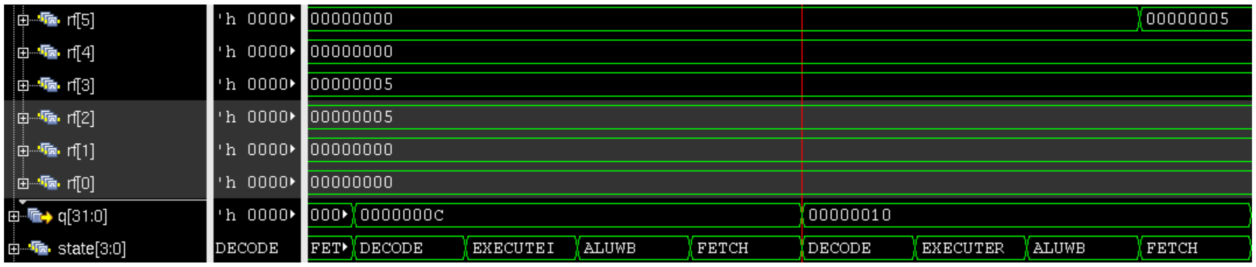

rf[5] 'h 0000 00000000 00000005
rf[4] 'h 0000 00000000
rf[3] 'h 0000 00000005
rf[2] 'h 0000 00000005
rf[1] 'h 0000 00000000
rf[0] 'h 0000 00000000
q[31:0] 'h 0000 000 0000000C 00000010
state[3:0] DECODE FET DECODE EXECUTEI ALUWB FETCH DECODE EXECUTER ALUWB FETCH

Figure 27: NOP operation followed by an OR operation

We can see that the NOP operation do absolutely nothing rather than increasing the program counter value. The OR operation between 5 and 5 is also 5 which is written after ALUWB state.

Acknowledgement

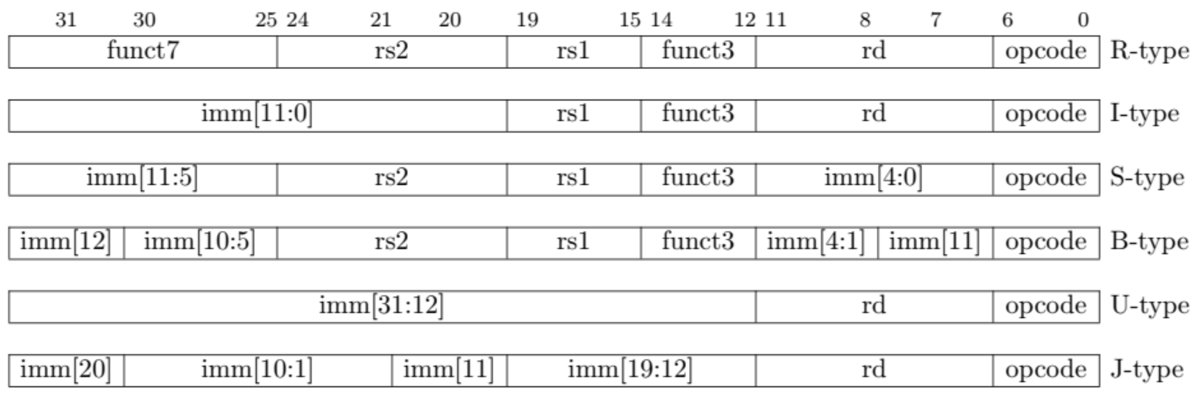
Our core design is adapted and edited from **Digital Design and Computer Architecture, RISC-V Edition: RISC-V Edition, chapter 7 exercises.**

We are also grateful to **Dr. A.B.M. Harun-Ur-Rashid Sir, Mumtahina Islam Sukanya madam** for guidance in the lab tasks and giving us the opportunity to do this project.

We should also thank the members of **NSL Team** for their constant support and regular meeting for different stages of design.

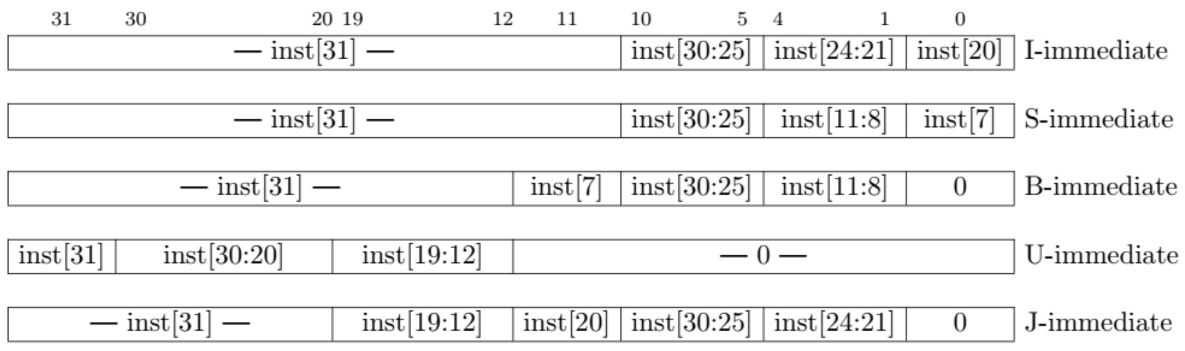
Reference

RISCV – 32i Specs Sheet:



31	30 25	24 21	20	19 15	14 12	11 8	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]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Figure 2.3: RISC-V base instruction formats showing immediate variants.

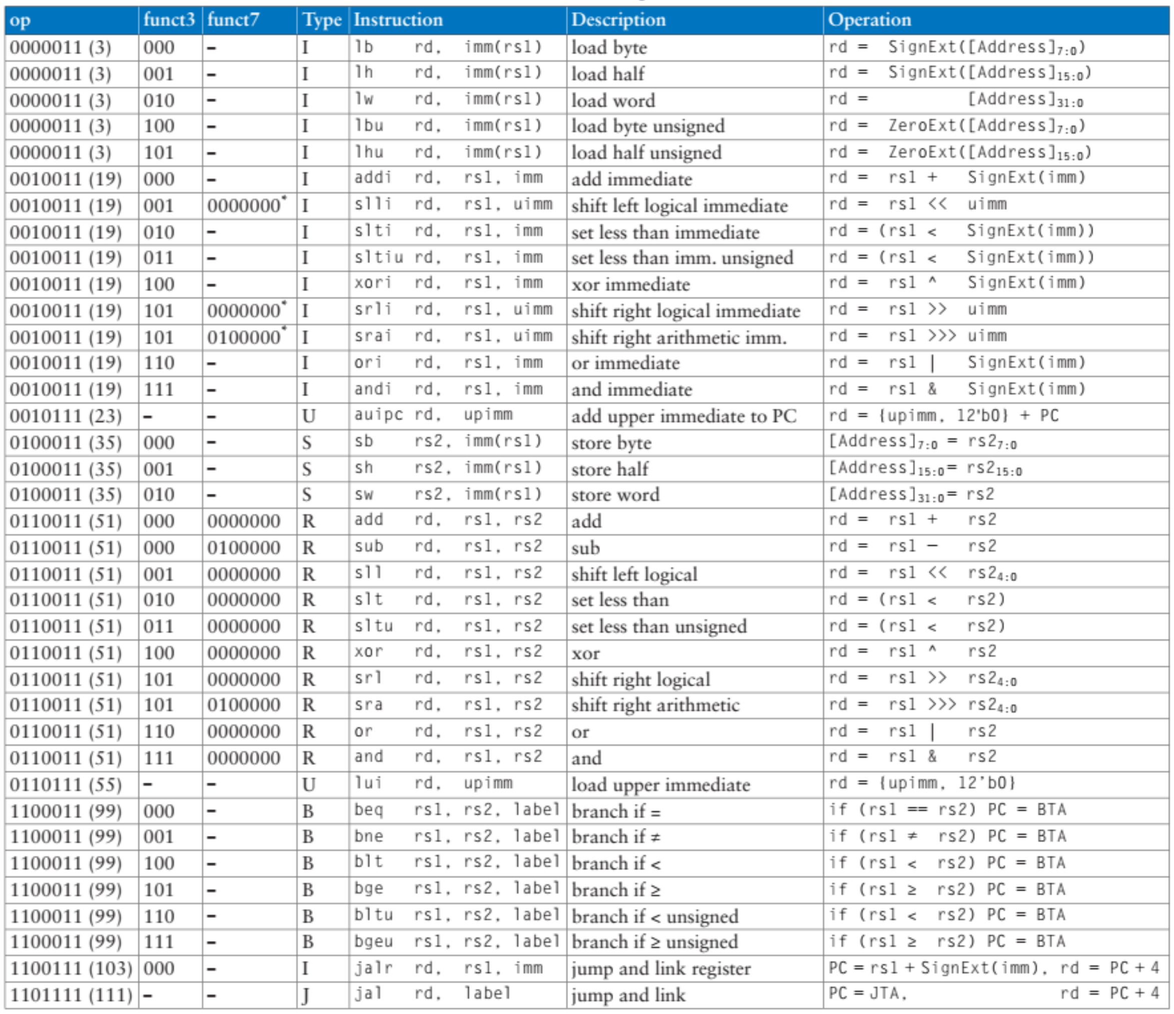


31	30 20	19 12	11	10 5	4 1	0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

Digital Design and Computer Architecture, RISC-V Edition: RISC-V Edition Appendix:

Table B.1 RV32I: RISC-V integer instructions



op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$= rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$= rs2
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 – rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

Conclusion

In this project we implemented a 32bit processor using RISCV architecture. As our memory is one integrated memory for both instruction and data, we implemented a multicycle processor with Von Neumann architecture.

The protocol for our task was to design APB protocol. We implemented an APB protocol but didn't get much time to integrate it to our processor. As a general protocol for APB was given, we discarded working on it and concentrated on core design.

We also wrote codes for synthesize our core but for more than one week the server was occupied with low storage and as a result we won't able to run our synthesis code to run and show the synthesized structure and further physical design.

Appendix:

Codes:

ALU:

```
module alu(
            input logic [31:0] a,
            input logic [31:0] b,
            input logic [3:0] alucontrol, // expanded to 4 bits for sra
            output logic [31:0] result,
            output logic [3:0] flags); // added for blt and other branches

            logic [31:0] condinvb, sum;
            logic v, c, n, z; // flags: overflow, carry out, negative, zero
            logic cout; // carry out of adder
            logic isAdd; // true if is an add operation
            logic isSub; // true if is a subtract operation

            assign flags = {v, c, n, z};
            assign condinvb = alucontrol[0] ? ~b : b;
            assign {cout, sum} = a + condinvb + alucontrol[0];
            assign isAddSub = ~alucontrol[3] & ~alucontrol[2] & ~alucontrol[1] | ~alucontrol[3] & ~alucontrol[1] &
alucontrol[0];
            always @(*) begin
                        case (alucontrol)
                                    4'b0000: result = sum; // add
                                    4'b0001: result = sum; // subtract
                                    4'b0010: result = a & b; // and
                                    4'b0011: result = a | b; // or
                                    4'b0100: result = a ^ b; // xor
                                    4'b0101: result = sum[31] ^ v; // slt
                                    4'b0110: result = a << b[4:0]; // sll
                                    4'b0111: result = a >> b[4:0]; // srl
                                    4'b1000: result = $signed(a) >>> b[4:0]; // sra
                                    default: result = 32'bx;
                        endcase
            end
            // added for blt and other branches
            assign z = (result == 32'b0);
            assign n = result[31];
            assign c = cout & isAddSub;
            assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;
endmodule
```

ALU Decoder:

```
module aludec(
            input logic opb5,
            input logic [2:0] funct3,
            input logic funct7b5,
            input logic [1:0] ALUOp,
            output logic [3:0] ALUControl); // expand to 4 bits for sra
```

```
        logic RtypeSub;
        assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract instruction

        always_comb
                case(ALUOp)
                        2'b00: ALUControl = 4'b000; // addition
                        2'b01: ALUControl = 4'b001; // subtraction
                        default: case(funct3) // R-type or I-type ALU
                                        3'b000: begin
                                                if (RtypeSub)
                                                        ALUControl = 4'b0001; // sub
                                                else
                                                        ALUControl = 4'b0000; // add, addi
                                        end
                                        3'b001: ALUControl = 4'b0110; // sll, slli
                                        3'b010: ALUControl = 4'b0101; // slt, slti
                                        3'b100: ALUControl = 4'b0100; // xor, xori
                                        3'b101: begin
                                                if (funct7b5)
                                                        ALUControl = 4'b1000; // sra, srai
                                                else
                                                        ALUControl = 4'b0111; // srl, srli
                                        end
                                        3'b110: ALUControl = 4'b0011; // or, ori
                                        3'b111: ALUControl = 4'b0010; // and, andi
                                        default: ALUControl = 4'bxxx;
                                endcase
                endcase
endmodule
```

Branching Unit:

```
module bu (
        input logic Branch,
        input logic [3:0] Flags,
        input logic [2:0] funct3,
        output logic taken);

        logic v, c, n, z; // Flags: overflow, carry out, negative, zero
        logic cond; // cond is 1 when condition for branch met

        assign {v, c, n, z} = Flags;
        assign taken = cond & Branch;

        always_comb begin
                case (funct3)
                        3'b000: cond = z; // beq
                        3'b001: cond = ~z; // bne
                        3'b100: cond = (n ^ v); // blt
                        3'b101: cond = ~(n ^ v); // bge
                        3'b110: cond = ~c; // bltu
```

```
                                3'b111: cond = c; // bgeu
                                default: cond = 1'b0;
                        endcase
                end
endmodule
```

Extending Unit:

```
module extend(
        input logic [31:7] instr,
        input logic [2:0] immsrc,
        output logic [31:0] immext);


        always_comb begin
                case(immsrc)
                        // I-type
                        3'b000: immext = {{20{instr[31]}}, instr[31:20]};
                        // S-type (stores)
                        3'b001: immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
                        // B-type (branches)
                        3'b010: immext = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};
                        // J-type (jal)
                        3'b011: immext = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0};
                        // U-type (auipc)
                        3'b100: immext = {instr[31:12], 12'b0};
                        default: immext = 32'bx; // undefined
                endcase
        end
endmodule
```

Flip Flop without Enable:

```
module flopr #(parameter WIDTH = 8)(
                        input logic clk, reset,
                        input logic [WIDTH-1:0] d,
                        output logic [WIDTH-1:0] q);

                        always_ff @(posedge clk, negedge reset) begin
                                if (!reset)
                                        q <= 0;
                                else
                                        q <= d;
                        end

endmodule
```

Flip Flop with Enable:

```
module flopenr #(parameter WIDTH = 8)(
        input logic clk, reset, en,
        input logic [WIDTH-1:0] d,
        output logic [WIDTH-1:0] q);

        always_ff @(posedge clk, negedge reset) begin
                if (!reset) q <= 0;
                else if (en) q <= d;
        end
endmodule
```

Instruction Decoder:

```
module instrdec (
        input logic [6:0] op,
        output logic [2:0] ImmSrc);

        always_comb begin
        case(op)
                7'b0110011:     ImmSrc = 3'bxxx; // R-type
                7'b0010011:     ImmSrc = 3'b000; // I-type ALU
                7'b0000011:     ImmSrc = 3'b000; // lw
                7'b0100011:     ImmSrc = 3'b001; // sw
                7'b1100011:     ImmSrc = 3'b010; // branches
                7'b1101111:     ImmSrc = 3'b011; // jal
                7'b1100111:     ImmSrc = 3'b000; // jalr
                7'b0010111:     ImmSrc = 3'b100; // auipc
                default:  ImmSrc = 3'bxxx;
        endcase
        end
endmodule
```

Load Store Unit:

```
module lsu(
        input logic [2:0] funct3,
        output logic LoadType, StoreType);
        always_comb
                case(funct3)
                        3'b000: {LoadType, StoreType} = 2'b01;
                        3'b010: {LoadType, StoreType} = 2'b00;
                        3'b100: {LoadType, StoreType} = 2'b1x;
                        default: {LoadType, StoreType} = 2'bxx;
                endcase
endmodule
```

2 by 1 MUX:

```
module mux2 #(parameter WIDTH = 8)(
        input logic [WIDTH-1:0] d0, d1,
        input logic s,
        output logic [WIDTH-1:0] y);

        assign y = s ? d1 : d0;
endmodule
```

3 by 1 MUX:

```
module mux3 #(parameter WIDTH = 8)(
        input logic [WIDTH-1:0] d0, d1, d2,
        input logic [1:0] s,
        output logic [WIDTH-1:0] y);

        assign y = s[1] ? d2 : (s[0] ? d1 : d0);

endmodule
```

Register File:

```
module regfile(
        input logic clk, reset,
        input logic we3,
        input logic [ 4:0] a1, a2, a3,
        input logic [31:0] wd3,
        output logic [31:0] rd1, rd2);


        logic [31:0] rf[31:0];
        // three ported register file
        // read two ports combinationally (A1/RD1, A2/RD2)
        // write third port on rising edge of clock (A3/WD3/WE3)
        // register 0 hardwired to 0
        always_ff @(posedge clk, negedge reset) begin
                if(!reset)
                        for (integer i=0; i<32; i=i+1)
                                rf[i] = 32'b0;
                else if(we3)
                        rf[a3] <= wd3;
        end

        assign rd1 = (a1 != 0) ? rf[a1] : 0;
        assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule
```

FSM for core control:

```
module mainfsm(
        input logic clk,
        input logic reset,
        input logic [6:0] op,
        output logic [1:0] ALUSrcA, ALUSrcB,
        output logic [1:0] ResultSrc,
        output logic AdrSrc,
        output logic IRWrite, PCUpdate,
        output logic RegWrite, MemWrite,
        output logic [1:0] ALUOp,
        output logic Branch);

        typedef enum logic [3:0] {
        FETCH, DECODE, MEMADR, MEMREAD,
        MEMWB, MEMWRITE,
        EXECUTER, EXECUTEI, ALUWB,
        BEQ, JAL
        } statetype;

        statetype state, nextstate;
        logic [14:0] controls;

        // state register
        always @(posedge clk or negedge reset) begin
            if (!reset)
                state <= FETCH;
            else
                state <= nextstate;
        end

        // next state logic
        always @(*)  begin
            case(state)
                FETCH: nextstate = DECODE;
                DECODE: casez(op)
                    7'b0?00011: nextstate = MEMADR; // lw or sw
                    7'b0110011: nextstate = EXECUTER; // R-type
                    7'b0010011: nextstate = EXECUTEI; // addi
                    7'b1100011: nextstate = BEQ; // beq
                    7'b1101111: nextstate = JAL; // jal
                    7'b1100111: nextstate = MEMADR; //jalr
                    7'b0010111: nextstate = ALUWB;    // auipc
                    default: nextstate = FETCH;   // if there is any anomaly or any incomplete
instruction
                endcase
                MEMADR: begin
                    casez(op[6:5])
                    2'b00:    nextstate = MEMREAD;     //memread
                    2'b01:    nextstate = MEMWRITE;   //memwrite
                    2'b11:    nextstate = JAL;   // jalr
                    endcase
                    end
                MEMREAD:  nextstate = MEMWB;
                EXECUTER: nextstate = ALUWB;
                EXECUTEI: nextstate = ALUWB;
                JAL:     nextstate = ALUWB;
                default: nextstate = FETCH;
```

```
                endcase
        end

        // state-dependent output logic
        always @(*)  begin
                case(state)
                        FETCH:          controls = 15'b00_10_10_0_1100_00_0;
                        DECODE:         controls = 15'b01_01_00_0_0000_00_0;
                        MEMADR:         controls = 15'b10_01_00_0_0000_00_0;
                        MEMREAD:        controls = 15'b00_00_00_1_0000_00_0;
                        MEMWRITE:       controls = 15'b00_00_00_1_0001_00_0;
                        MEMWB:          controls = 15'b00_00_01_0_0010_00_0;
                        EXECUTER:       controls = 15'b10_00_00_0_0000_10_0;
                        EXECUTEI:       controls = 15'b10_01_00_0_0000_10_0;
                        ALUWB:          controls = 15'b00_00_00_0_0010_00_0;
                        BEQ:            controls = 15'b10_00_00_0_0000_01_1;
                        JAL:            controls = 15'b01_10_00_0_0100_00_0;
                        default:  controls = 15'bxx_xx_xx_x_xxxx_xx_x;
                endcase
        end

assign {ALUSrcA, ALUSrcB, ResultSrc, AdrSrc, IRWrite, PCUpdate,       RegWrite,MemWrite, ALUOp, Branch} =
controls;
endmodule
```

Controller:

```
module controller(
        input logic clk,
        input logic reset,
        input logic [6:0] op,
        input logic [2:0] funct3,
        input logic funct7b5,
        input logic [3:0] Flags,
        output logic [2:0] ImmSrc,
        output logic [1:0] ALUSrcA, ALUSrcB,
        output logic [1:0] ResultSrc,
        output logic AdrSrc,
        output logic [3:0] ALUControl,
        output logic IRWrite, PCWrite,
        output logic RegWrite, MemWrite,
        output logic LoadType,
        output logic StoreType,
        output logic PCTargetSrc,
        output logic mem_en,
        output logic [1:0] mem_data_length);


        logic [1:0] ALUOp;
        logic Branch, PCUpdate;
        logic branchtaken; // added for other branches
```

```
	// Main FSM
	mainfsm fsm(
		clk,
		reset,
		op,
		ALUSrcA,
		ALUSrcB,
		ResultSrc,
		AdrSrc,
		IRWrite,
		PCUpdate,
		RegWrite,
		MemWrite,
		ALUOp,
		Branch);

	// ALU Decoder
	aludec ad(
		op[5],
		funct3,
		funct7b5,
		ALUOp,
		ALUControl);

	// Instruction Decoder
	instrdec id(
		op,
		ImmSrc);

	// Branch logic
	lsu lsu(
		funct3,
		LoadType,
		StoreType);

	bu branchunit(
		Branch,
		Flags,
		funct3,
		branchtaken); // added for bne,	blt, etc.

	assign PCWrite = branchtaken | PCUpdate;

	assign mem_en = MemWrite | !MemWrite;
	assign mem_data_length = 2'b00;
endmodule
```

Datapath:

```
module datapath(
        input logic clk, reset,
        input logic [2:0] ImmSrc,
        input logic [1:0] ALUSrcA, ALUSrcB,
        input logic [1:0] ResultSrc,
        input logic AdrSrc,
        input logic IRWrite, PCWrite,
        input logic RegWrite, MemWrite,
        input logic [3:0] alucontrol,
        input logic LoadType, StoreType,
        input logic PCTargetSrc,
        output logic [6:0] op,
        output logic [2:0] funct3,
        output logic funct7b5,
        output logic [3:0] Flags,
        output logic [31:0] Adr,
        input logic [31:0] ReadData,
        output logic [31:0] WriteData
        );

        logic [31:0] PC, OldPC, Instr, immext, ALUResult;
        logic [31:0] SrcA, SrcB, RD1, RD2, A;
        logic [31:0] Result, Data, ALUOut;


        // next PC logic
        flopenr #(32) pcreg(clk, reset, PCWrite, Result, PC);
        flopenr #(32) oldpcreg(clk, reset, IRWrite, PC, OldPC);

        // memory logic
        mux2 #(32) adrmux(PC, Result, AdrSrc, Adr);
        flopenr #(32) ir(clk, reset, IRWrite, ReadData, Instr);
        flopr #(32) datareg(clk, reset, ReadData, Data);

        // register file logic
        regfile rf(
                clk, reset,
                RegWrite,
                Instr[19:15],
                Instr[24:20],
                Instr[11:7],
                Result,
                RD1,
                RD2);

        extend ext(
                Instr[31:7],
                ImmSrc,
                immext);

        flopr #(32) srcareg(clk, reset, RD1, A);
        flopr #(32) wdreg(clk, reset, RD2, WriteData);
        // ALU logic
        mux3 #(32) srcamux(PC, OldPC, A, ALUSrcA, SrcA);
        mux3 #(32) srcbmux(WriteData, immext, 32'd4, ALUSrcB, SrcB);
        alu alu(SrcA, SrcB, alucontrol, ALUResult, Flags);
        flopr #(32) aluoutreg(clk, reset, ALUResult, ALUOut);
        mux3 #(32) resmux(ALUOut, Data, ALUResult, ResultSrc, Result);

        // outputs to control unit
        assign op = Instr[6:0];
        assign funct3 = Instr[14:12];
        assign funct7b5 = Instr[30];
endmodule
```

Processor with instantiation:

```
module riscvmulti(
        input logic clk, reset,
        output logic MemWrite,
        output logic [31:0] Adr, WriteData,
        output logic mem_en,
        output logic [1:0] mem_data_length,
        input logic [31:0] ReadData);

        logic RegWrite, jump;
        logic [1:0] ResultSrc;
        logic [2:0] ImmSrc; // expand to 3-bits for auipc
        logic [3:0] ALUControl;
        logic PCWrite;
        logic IRWrite;
        logic [1:0] ALUSrcA;
        logic [1:0] ALUSrcB;
        logic AdrSrc;
        logic [3:0] Flags; // added for other branches
        logic [6:0] op;
        logic [2:0] funct3;
        logic funct7b5;
        logic LoadType;
        logic StoreType;
        logic PCTargetSrc; // added for jalr

        controller c(
                clk,
                reset,
                op,
                funct3,
                funct7b5,
                Flags,
                ImmSrc,
                ALUSrcA,
                ALUSrcB,
                ResultSrc,
                AdrSrc,
                ALUControl,
                IRWrite,
                PCWrite,
                RegWrite,
                MemWrite,
                LoadType,
                StoreType,
                PCTargetSrc,
                mem_en,
                mem_data_length);

        datapath dp(
                clk,
                reset,
                ImmSrc,
                ALUSrcA,
                ALUSrcB,
                ResultSrc,
                AdrSrc,
                IRWrite,
                PCWrite,
                RegWrite,
                MemWrite,
                ALUControl,
                LoadType,
                StoreType,
                PCTargetSrc,
                op,
                funct3,
                funct7b5,
                Flags,
                Adr,
                ReadData,
                WriteData
                );
endmodule
```

Processor with integration to memory:

```
module rv32i_pipelined_top #(parameter DATA_LENGTH = 32, parameter ADDRESS_LENGTH = 32) (
clk,rst_n,run_complete,core_select,addr_in,data_in,instruction_load_start,pselect,pwrite,pready, data_out);

   input clk;
   input rst_n;
   input core_select;

   //apb signals
   input [31:0] addr_in;
   input [31:0] data_in;
   input pselect;
   input pwrite;
   input pready;

   input instruction_load_start;

   output wire [31:0] data_out;
   output wire run_complete;


wire
from_apb_mem_wr_en_wire,from_apb_mem_rd_en_wire,from_core_to_imem_en_wire,from_core_to_imem
_wr_en_wire,from_core_to_imem_rd_en_wire,from_core_to_dmem_en_wire,from_core_to_dmem_wr_en_wi
re,from_core_to_dmem_rd_en_wire,from_top_to_spi_mosi_in,from_top_to_spi_miso_out,to_inst_mem_en_w
ire,to_inst_mem_wr_en_wire,to_inst_mem_rd_en_wire,to_data_mem_en_wire,to_data_mem_wr_en_wire,to
_data_mem_rd_en_wire,from_spi_mem_en_wire,from_apb_mem_en_wire,from_spi_mem_wr_en_wire,from_
spi_mem_rd_en_wire,to_apb_pwrite,to_apb_pready,to_apb_psel;

wire [DATA_LENGTH-1:0]
from_apb_mem_address_wire,from_apb_mem_data_in_wire,from_apb_mem_data_out_wire,from_top_to_ap
b_out,from_core_to_imem_address_wire,from_core_to_imem_data_in_wire,from_core_to_dmem_address_w
ire,from_core_to_dmem_data_in_wire,from_imem_to_core_data_wire,from_spi_mem_address_wire,from_spi
_mem_data_in_wire,from_spi_mem_data_out_wire,from_dmem_to_core_data_wire,to_data_mem_address_
wire,to_data_mem_data_in_wire,to_inst_mem_address_wire,to_inst_mem_data_in_wire,from_top_to_apb_a
ddr_in,from_top_to_apb_data_in;

wire                                                                                  [1:0]
from_apb_mem_data_length_wire,from_spi_mem_data_length_wire,from_core_to_dmem_data_length_wire,f
rom_core_to_imem_data_length_wire,to_inst_mem_data_length_wire,to_data_mem_data_length_wire;


assign from_top_to_apb_addr_in = addr_in;
assign from_top_to_apb_data_in = data_in;
assign data_out = from_top_to_apb_out;

assign to_apb_pready = pready;
assign to_apb_pwrite = pwrite;
assign to_apb_psel = pselect;


assign to_inst_mem_en_wire = (instruction_load_start) ? from_apb_mem_en_wire : 1'b0;
assign to_inst_mem_wr_en_wire = (instruction_load_start) ? from_apb_mem_wr_en_wire : 1'b0;
assign to_inst_mem_rd_en_wire = (instruction_load_start) ? from_apb_mem_rd_en_wire : 1'b0;
assign to_inst_mem_address_wire = (instruction_load_start) ? from_apb_mem_address_wire : {{DATA_LENGTH-
1}{1'b0}};
assign to_inst_mem_data_in_wire = (instruction_load_start) ? from_apb_mem_data_in_wire : {{DATA_LENGTH-
1}{1'b0}};
assign to_inst_mem_data_length_wire = (instruction_load_start) ? from_apb_mem_data_length_wire : 2'b0;

wire core_reset_n;

assign core_reset_n = core_select & rst_n;
assign from_core_to_imem_rd_en_wire = !from_core_to_imem_wr_en_wire;
```

```
riscvmulti riscvmulti(
        .clk(clk),
        .reset(core_reset_n),
               .MemWrite(from_core_to_imem_wr_en_wire),
               .Adr(from_core_to_imem_address_wire),
               .WriteData(from_core_to_imem_data_in_wire),
        .mem_en(from_core_to_imem_en_wire),
               .mem_data_length(from_core_to_imem_data_length_wire),


               .ReadData(from_imem_to_core_data_wire)
 );

data_memory_wrapper #(DATA_LENGTH,ADDRESS_LENGTH) imem_wrapper (
    .clk(clk),
    .core_select(core_select),

    .from_core_mem_en(from_core_to_imem_en_wire),
    .from_core_mem_wr_en(from_core_to_imem_wr_en_wire),
    .from_core_mem_rd_en(from_core_to_imem_rd_en_wire),
    .from_core_mem_address(from_core_to_imem_address_wire),
    .from_core_mem_data_in(from_core_to_imem_data_in_wire),
    .from_core_mem_data_length(from_core_to_imem_data_length_wire),

    .from_intf_mem_ctrl_mem_en(to_inst_mem_en_wire),
    .from_intf_mem_ctrl_mem_wr_en(to_inst_mem_wr_en_wire),
    .from_intf_mem_ctrl_mem_rd_en(to_inst_mem_rd_en_wire),
    .from_intf_mem_ctrl_mem_address(to_inst_mem_address_wire),
    .from_intf_mem_ctrl_mem_data_in(to_inst_mem_data_in_wire),
    .from_intf_mem_ctrl_mem_data_length(to_inst_mem_data_length_wire),

    .to_core_mem_data_out(from_imem_to_core_data_wire),
    .to_intf_mem_ctrl_mem_data_out(from_apb_mem_data_out_wire)
);


apb_slave #(DATA_LENGTH,ADDRESS_LENGTH) apb_mem(
    .from_top_clk(clk),
    .preset_n(rst_n),
    .pwrite(to_apb_pwrite),
    .psel(to_apb_psel),
    .pready(to_apb_pready),

    .from_top_apb_paddr(from_top_to_apb_addr_in),
    .from_top_apb_pwdata(from_top_to_apb_data_in),

    .prdata(from_top_to_apb_out),

    .to_mem_en(from_apb_mem_en_wire),
    .to_mem_wr_en(from_apb_mem_wr_en_wire),
    .to_mem_rd_en(from_apb_mem_rd_en_wire),
    .to_mem_address(from_apb_mem_address_wire),
    .to_mem_data_in(from_apb_mem_data_in_wire),
   .to_mem_data_length(from_apb_mem_data_length_wire),

   .from_mem_data_out(from_apb_mem_data_out_wire)
);
endmodule
```

APB:

```
module apb_slave #(
   parameter DATA_LENGTH = 32,ADDRESS_LENGTH = 32
   ) (
      from_top_clk, preset_n,
      from_top_apb_paddr, pwrite,
      psel, pready,
      from_top_apb_pwdata,
      to_mem_en,
      to_mem_wr_en,
      to_mem_rd_en,
      to_mem_address,
      to_mem_data_in,
      to_mem_data_length,
      from_mem_data_out,prdata
   );

   input                  from_top_clk, preset_n, pwrite, psel, pready;
   input [DATA_LENGTH-1:0]     from_top_apb_paddr; //paddr
   input [DATA_LENGTH-1:0]     from_top_apb_pwdata; //pwdata

   // mem_wrapper_signals
   input [DATA_LENGTH-1:0]     from_mem_data_out;
   output                  to_mem_en, to_mem_wr_en, to_mem_rd_en;
   output [DATA_LENGTH-1:0]    to_mem_address;
   output [DATA_LENGTH-1:0]    to_mem_data_in;
   output [1:0]             to_mem_data_length;

   output reg [DATA_LENGTH-1:0]    prdata;

   reg [2:0] apb_state;
   reg penable;

   reg [DATA_LENGTH-1:0]     apb_data_in;
   reg [DATA_LENGTH-1:0]     apb_address_in;

   parameter IDLE         = 3'b000;
   parameter SETUP        = 3'b001;
   //parameter WRITE_DATA = 2'b10;
   parameter WRITE_EN      = 3'b010;
   parameter READ_EN       = 3'b011;
   parameter DONE          = 3'b100;

   assign to_mem_wr_en      = (apb_state == DONE) ? 1'b1 : 1'b0;
```

```
  assign to_mem_rd_en       = (to_mem_wr_en) ? 1'b0 : 1'b1;
  assign to_mem_en          = (apb_state == DONE) ? 1'b1 : (1'b0 | to_mem_rd_en);
  assign to_mem_address     = (apb_state == DONE) ? apb_address_in : to_mem_address;
  assign to_mem_data_length = 2'b11;
  assign to_mem_data_in     =  (apb_state == DONE) ? apb_data_in : to_mem_data_in;

always @(posedge from_top_clk) begin
  if (~preset_n) begin
    apb_address_in     <= {DATA_LENGTH{1'b0}};
    apb_data_in        <= {DATA_LENGTH{1'b0}};
    apb_state          <= IDLE;
    penable <= 1'b1;
  end

  else begin
    case (apb_state)
      IDLE: begin
        if (psel) begin
          apb_state <= SETUP;
        end
      end

      SETUP : begin
        if (psel && penable) begin
          if (pwrite) begin
            penable = 1'b1;
            apb_state <= WRITE_EN;
          end
          else begin
            penable = 1'b1;
            apb_state <= READ_EN;
          end
        end
      end

      WRITE_EN : begin
        if (psel && penable && pwrite && pready) begin
          apb_address_in <= from_top_apb_paddr;
          apb_data_in <= from_top_apb_pwdata;
          apb_state <= DONE;
        end
      end

      READ_EN : begin
        if (psel && penable && !pwrite && pready) begin
          apb_address_in <= from_top_apb_paddr;
          prdata <= from_mem_data_out;
          apb_state <= DONE;
        end
      end

      DONE : begin
        penable <= 1'b1;
```

```
            apb_state <= IDLE;
          end
          default: begin
            penable <= 1'b1;
            apb_state <= IDLE;
          end
        endcase
    end
end
endmodule
```

Memory Wrapper:

```
module data_memory_wrapper #( parameter DATA_LENGTH=32, ADDRESS_LENGTH=32) (clk, core_select,
        from_core_mem_en, from_core_mem_wr_en, from_core_mem_rd_en, from_core_mem_address,
        from_core_mem_data_in, from_core_mem_data_length, to_core_mem_data_out,


        from_intf_mem_ctrl_mem_en, from_intf_mem_ctrl_mem_wr_en, from_intf_mem_ctrl_mem_rd_en,
        from_intf_mem_ctrl_mem_address, from_intf_mem_ctrl_mem_data_in,
        from_intf_mem_ctrl_mem_data_length, to_intf_mem_ctrl_mem_data_out);


        input clk;
        input core_select;

        //core
        input from_core_mem_en;
        input from_core_mem_wr_en;
        input from_core_mem_rd_en;
        input [DATA_LENGTH-1:0] from_core_mem_address;
        input [DATA_LENGTH-1:0] from_core_mem_data_in;
        input [1:0] from_core_mem_data_length;
        output wire [DATA_LENGTH-1:0] to_core_mem_data_out;

        //top
        input from_intf_mem_ctrl_mem_en;
        input from_intf_mem_ctrl_mem_wr_en;
        input from_intf_mem_ctrl_mem_rd_en;
        input [DATA_LENGTH-1:0] from_intf_mem_ctrl_mem_address;
        input [DATA_LENGTH-1:0] from_intf_mem_ctrl_mem_data_in;
        input [1:0] from_intf_mem_ctrl_mem_data_length;
        output wire [DATA_LENGTH-1:0] to_intf_mem_ctrl_mem_data_out;
        wire mem_wr_en, mem_en, mem_rd_en;
        wire [DATA_LENGTH-1:0] mem_data;
        wire [ADDRESS_LENGTH-1:0] mem_address;

        assign mem_en = core_select ? from_core_mem_en: from_intf_mem_ctrl_mem_en;
        assign mem_wr_en = core_select ? from_core_mem_wr_en: from_intf_mem_ctrl_mem_wr_en;
        assign mem_rd_en = core_select ? from_core_mem_rd_en: from_intf_mem_ctrl_mem_rd_en;
```

```
	assign mem_data = core_select ? from_core_mem_data_in : from_intf_mem_ctrl_mem_data_in;
	assign mem_address = core_select ? from_core_mem_address[ADDRESS_LENGTH-1:2]:from_intf_mem_ctrl_mem_address[ADDRESS_LENGTH-1:0];

	wire [3:0] mem_we_out;
	wire [3:0] spi_mem_we_out;
	wire [3:0] core_mem_we_out;
	wire [DATA_LENGTH-1:0] mem_data_out_wire;

	assign spi_mem_we_out = (from_intf_mem_ctrl_mem_data_length==2'b00) ? 4'b0000 :
	(from_intf_mem_ctrl_mem_data_length==2'b01) ? 4'b0001: (from_intf_mem_ctrl_mem_data_length==2'b10) ? 4'b0011 : 4'b1111;
	assign core_mem_we_out = (from_core_mem_data_length==2'b00) ? 4'b1111 : (from_core_mem_data_length==2'b01) ? 4'b0011 :(from_core_mem_data_length==2'b10) ? 4'b0001 :4'b0000;
	assign mem_we_out = mem_wr_en ? core_select ? core_mem_we_out : spi_mem_we_out : 4'b0000;
	assign to_core_mem_data_out = (core_select & mem_rd_en) ? mem_data_out_wire :to_core_mem_data_out;
	assign to_intf_mem_ctrl_mem_data_out = (~core_select & mem_rd_en) ? mem_data_out_wire : to_intf_mem_ctrl_mem_data_out;


	DFFRAM_RTL_2048 #(ADDRESS_LENGTH,DATA_LENGTH) memory_from_DFFRAM (
		.CLK(clk),
		.WE(mem_we_out),
		.EN(mem_en),
		.Di(mem_data),
		.Do(mem_data_out_wire),
		.A(mem_address)
	);

endmodule
```

Ram:

```
module DFFRAM_RTL_2048 #(parameter ADDRESS_LENGTH=32, parameter DATA_LENGTH=32)(
	CLK,
	WE,
	EN,
	Di,
	Do,
	A
);
	input wire CLK;
	input wire [3:0] WE;
	input wire EN;
	input wire [(DATA_LENGTH) -1:0] Di;
	output [(DATA_LENGTH) -1:0] Do;
	input wire [(ADDRESS_LENGTH - 1): 0] A;
	reg [(DATA_LENGTH) -1:0] RAM[2047 : 0];

	assign Do = EN ? (~WE ? RAM[A] : Do) : 32'b0;
```

```
        always @(posedge CLK)
                if(EN) begin
                        if(WE[0]) RAM[A][ 7: 0] <= Di[7:0];
                        if(WE[1]) RAM[A][15: 8] <= Di[15:8];
                        if(WE[2]) RAM[A][23:16] <= Di[23:16];
                        if(WE[3]) RAM[A][31:24] <= Di[31:24];
                end
endmodule
```

Top Testbench Module:

```
module rv32i_top_tb #(parameter DATA_LENGTH = 32,ADDRESS_LENGTH = 11) ();



        int k;
        string line;
        int fd;



        logic instruction_load_start;
        logic clk;
        logic rst_n;
        logic core_select;

        logic [31:0] addr_in;
  logic [31:0] data_in;
        logic pselect;
        logic pwrite;
        logic pready;

        logic run_complete;
        logic [31:0] data_out;

        // variable for address writeout to memory through apb
        logic [ADDRESS_LENGTH -1:0] to_apb_address_out;
        // variable for data writeout to memory through apb
        logic [DATA_LENGTH -1:0] to_apb_data_out;


        int j = 0;



        rv32i_pipelined_top #(DATA_LENGTH,ADDRESS_LENGTH) rv32i_pipelined_top(
                .clk(clk),
                .rst_n(rst_n),
                .core_select(core_select),
                .addr_in(addr_in),
                .data_in(data_in),
                .pselect(pselect),
                .pwrite(pwrite),
                .pready(pready),
                .data_out(data_out),
                .run_complete(run_complete),
                .instruction_load_start(instruction_load_start)
        );


        initial begin
                clk = 0;
                core_select = 0;
                rst_n = 1;
                pselect = 1'b0;
                pready = 1'b0;
                pwrite = 1'b0;
                addr_in = 32'b0;
                data_in = 32'b0;

                instruction_load_start =1'b0;

                reset_event();


                instruction_load_start = 1'b1;
                write_instructions();


                run_core();
                #10 $finish;
        end

        always @ (posedge clk) begin
    if (run_complete) begin
      #10;
      $finish;
    end
  end

        initial begin
                $dumpfile("dump.vcd");
                $dumpvars;
                #30000 $finish();
        end

        always #5 clk = ~clk;

        task reset_event();
                // reset APB registers
                repeat(1) @(posedge clk);
                rst_n = 1'b0;
                repeat(2) @(posedge clk);
                rst_n = 1'b1;
        endtask

        task write_instructions();
		logic [DATA_LENGTH -1:0] mem [2047:0];
		$readmemh("prime_test.txt", mem);///pipelined_top

		line_number_detection();

		for (int i=0; i<=k; i++) begin
			to_apb_address_out = i;
			to_apb_data_out = mem[i];
			apb_write();
		end

		instruction_load_start = 1'b0;
	endtask


	task apb_write();
		addr_in = to_apb_address_out;

		data_in = to_apb_data_out;


		pselect = 1'b1;
		pwrite = 1'b1;
		pready = 1'b1;
		repeat(4) @(posedge clk);

		pselect = 1'b0;

	endtask


	task write_data();
		logic [DATA_LENGTH -1:0] mem [2047:0];
		$readmemh("pixel_exec.txt", mem);///pipelined_top

		line_number_detection_2();

		for (int i=660; i<=660+k*4; i=i+4) begin
			to_apb_address_out = i;
			to_apb_data_out = mem[0+j];
			j=j+1;
		end

		instruction_load_start = 1'b1;
	endtask



	task run_core();
		core_select = 1'b1;
		rst_n = 1'b0;
		repeat(2) @(posedge clk);
		rst_n = 1'b1;
```

```
			repeat(1) @(posedge clk);
		endtask

		task line_number_detection();

			k=0;
			fd=$fopen("prime_test.txt","r");

			while( !$feof(fd) ) begin
				integer code;
				code = $fgets(line,fd);
				k=k+1;
			end
			$fclose(fd);

		endtask

		task line_number_detection_2();

			k=0;
			fd=$fopen("./pipelined_top/TB/img_px.txt","r");

			while( !$feof(fd) ) begin
				integer code;
				code = $fgets(line,fd);
				k=k+1;
			end
			$fclose(fd);

		endtask
endmodule
```

Synthesis File (Couldn’t be run for system storage shortage)

```
##rtl.tcl file adapted from http://ece.colorado.edu/~ecen5007/cadence/
##this tells the compiler where to look or the libraries


set_attribute lib_search_path /home/vlsi05/library

## This defines the libraries to use

set_attribute library {slow_vdd1v0_basicCells.lib fast_vdd1v0_basicCells.lib}

##This must point to your VHDL/verilog file


read_hdl -sv alu.sv
read_hdl -sv aludec.sv
read_hdl -sv bu.sv
read_hdl -sv extend.sv
read_hdl -sv instrdec.sv
read_hdl -sv regfile.sv
read_hdl -sv flopr.sv
read_hdl -sv flopenr.sv
read_hdl -sv lsu.sv
read_hdl -sv mux2.sv
read_hdl -sv mux3.sv
read_hdl -sv mainfsm.sv
read_hdl -sv controller.sv
read_hdl -sv datapath.sv
read_hdl -sv riscvmulti.sv


## This builts the general block
elaborate

##this allows you to define a clock and the maximum allowable delays
## READ MORE ABOUT THIS SO THAT YOU CAN PROPERLY CREATE A TIMING FILE
#set clock [define_clock -period 300 -name clk]
#external delay -input 300 -edge rise clk
#external delay -output 2000 -edge rise p1

##This synthesizes your code
synthesize -to_mapped

## This writes all your files
## change the tst to the name of your top level verilog
## CHANGE THIS LINE: CHANGE THE "accu" PART REMEMBER THIS
## FILENAME YOU WILL NEED IT WHEN SETTING UP THE PLACE & ROUTE
write -mapped > synth_codes/core_synth.v

## THESE FILES ARE NOT REQUIRED, THE SDC FILE IS A TIMING FILE
write_script > script
```