

KGP-RISC Documentation

Implementation of a Processor in Verilog

Lovish Chopra 16CS10058

Himanshu Mundhra 16CS10057

Group 56

Instruction Set Architecture (ISA)

Class	Instruction	Usage	Meaning
Arithmetic	Add	add rs,rt	$rs \leftarrow (rs) + (rt)$
	Comp	comp rs,rt	$rs \leftarrow 2\text{'s Complement } (rt)$
	Add immediate	addi rs,imm	$rs \leftarrow (rs) + imm$
	Complement Immediate	compi rs,imm	$rs \leftarrow 2\text{'s Complement } (imm)$
Logic	AND	and rs,rt	$rs \leftarrow (rs) \wedge (rt)$
	XOR	xor rs,rt	$rs \leftarrow (rs) \oplus (rt)$
Shift	Shift left logical	shll rs, sh	$rs \leftarrow (rs)$ left-shifted by sh
	Shift right logical	shrl rs, sh	$rs \leftarrow (rs)$ right-shifted by sh
	Shift left logical variable	shllv rs, rt	$rs \leftarrow (rs)$ left-shifted by (rt)
	Shift right logical variable	shrlv rs, rt	$rs \leftarrow (rs)$ right-shifted by (rt)
	Shift right arithmetic	shra rs, sh	$rs \leftarrow (rs)$ arithmetic right-shifted by sh
	Shift right arithmetic variable	shrav rs, rt	$rs \leftarrow (rs)$ right-shifted by (rt)
Memory	Load Word	lw rt,imm(rs)	$rt \leftarrow mem[(rs) + imm]$
	Store Word	sw rt,imm,(rs)	$mem[(rs) + imm] \leftarrow (rt)$
Branch	Unconditional branch	b L	goto L
	Branch Register	br rs	goto (rs)
	Branch on zero	bz L	if ($zflag == 1$) then goto L
	Branch on not zero	bnz L	if ($zflag == 0$) then goto L
	Branch on Carry	bcy L	if ($carryflag == 1$) then goto L
	Branch on No Carry	bncy L	if ($carryflag == 0$) then goto L
	Branch on Sign	bs	if ($signflag == 1$) then goto L
	Branch on Not Sign	bns	if ($signflag == 0$) then goto L
	Branch on Overflow	bv	if ($overflowflag == 1$) then goto L
	Branch on No Overflow	bnv	if ($overflowflag == 0$) then goto L
	Call	Call L	$r31 \leftarrow (PC)+4$; goto L
	Return	Ret	goto (ra)

Register Usage Convention:

Register	Function	Register Number	Register Code
\$zero	0 register, stores the constant 0	0	00000
\$v0-\$v1	Saved variable	1-2	00001-00010
\$a0-\$a3	Parameters for a function call	3-6	00011-00110
\$t0-\$t12	Temporaries	7-19	00111-10011
\$s0-\$s10	Saved variables, preserved during	20-30	10100-11110

	function calls		
\$ra	Return address	31	11111

Instruction Format and Encoding:

The various instructions in the KGP-RISC ISA can be categorised into the following five categories:

Opcode	Binary Representation	Functions
0	000	xor, and, comp, add, shll, shrl, shllv, shrlv, shra, shrav
1	001	compi, addi
2	010	lw, sw
3	011	b, bz, bnz, bcy, bncy, bs, bns, bv, bnv, Call, Ret
4	100	br

Opcode: 000

Format:

Opcode	rs	rt	shamt	func	Don't Care
3 bits	5 bits	5 bits	5 bits	4 bits	10 bits

Functions:

Function	Function Code	Binary Representation
xor	0	0000
and	1	0001
comp	2	0010
add	3	0011
shll	4	0100
shrl	5	0101
shllv	6	0110
shrlv	7	0111

shra	8	1000
shrav	9	1001

Opcode: 001

Format:

Opcode	rs	imm	func
3 bits	5 bits	22 bits	2 bits

Functions:

Function	Function Code	Binary Representation
compi	0	00
addi	1	01

Opcode: 010

Format:

Opcode	rs	rt	imm	func
3 bits	5 bits	5 bits	18 bits	1 bit

Functions:

Function	Function Code	Binary Representation
lw	0	0
sw	1	1

Opcode: 011

Format:

Opcode	L	func
3 bits	25 bits	4 bits

Functions:

Function	Function Code	Binary Representation
b	0	0000

bz	1	0001
bnz	2	0010
bcy	3	0011
bncy	4	0100
bs	5	0101
bns	6	0110
bv	7	0111
bnv	8	1000
Call	9	1001
Ret	10	1010

Opcode: 100

Format:

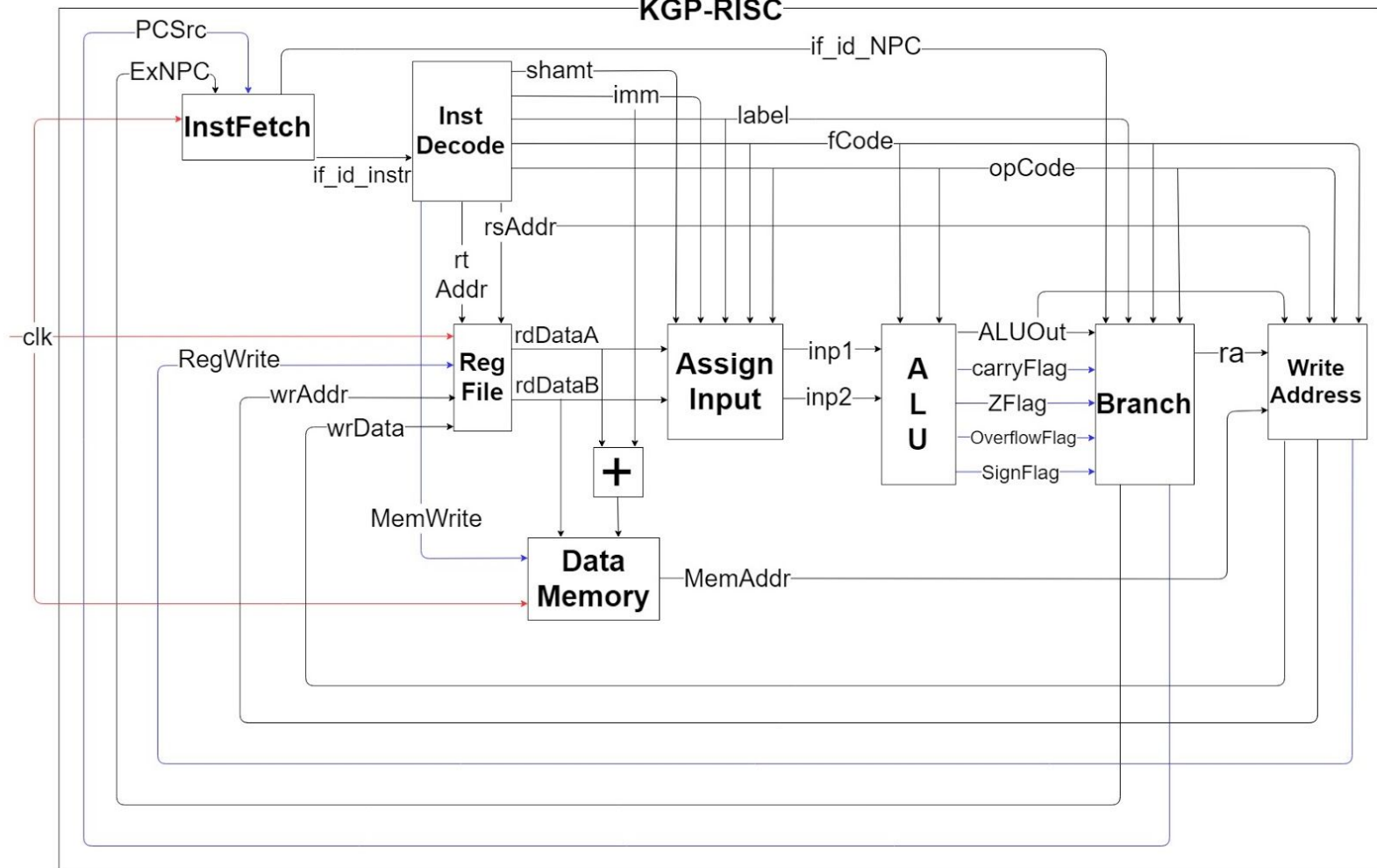
Opcode	rs	Don't Care
3 bits	5 bits	24 bits

Functions:

br

Architecture

KGP-RISC



Flags: (marked as blue in the Architecture)

- carryFlag:** is set to 1 when an operation generates a carry, *otherwise zero*.
- zFlag:** is set to 1 when an operation generates a zero output, *otherwise zero*.
- overflowFlag:** is set to 1 when an operation generates an overflow, *otherwise zero*.
- signFlag:** is set to 1 when an operation generates a negative output, *otherwise zero*.
- MemWrite:** is set to 1 on store_word (sw), *otherwise zero*.
- RegWrite:** is set to 1 when we need to write into a register, *otherwise zero*.
- PCSrc:** is set to 1 when a branch operation is encountered, *otherwise zero*.

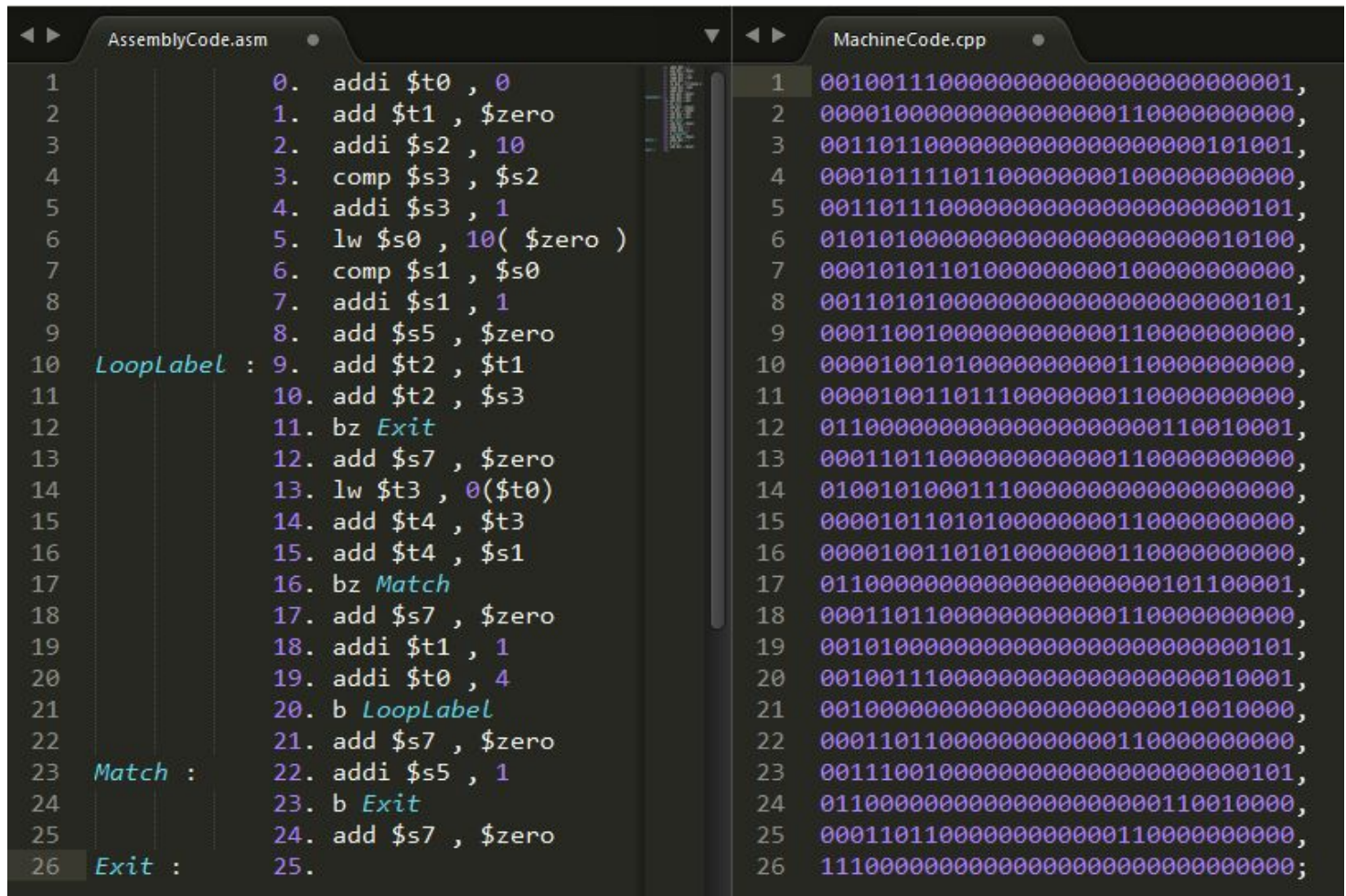
Modules and their Purpose:

1. **Instruction Fetch Module**: Used to fetch the instruction from the BRAM based on the PC. It is comprised of 5 parts:
 - a. Mux: To select between PC+1 and ExNPC
 - b. PC: Assigns the output of mux to PC at every clock cycle
 - c. Instruction_Memory: Block RAM (Single Port ROM), Instruction is fetched from here
 - d. IF_ID: Used to assign the values of PC+1 and Instruction for later use in Processor
 - e. PC_Incrementor: Gives as output PC+1
2. **Instruction Decoder Module**: Splits the instruction into opcode, fcode, rsAddress, rtAddress, imm, label and shamt. Also, it decides if we need to write anything in Data Memory or not using MemWrite.
3. **RegFile32x32** : Register file, used to write into and read from registers. Allows two registers to be read at a time.
4. **assignInputs**: Based on the opcode and fcode, this modules assigns what two inputs will be present in the ALU. for example, in arithmetic instructions, it is rs and rt whereas in shift instructions it is rs and shamt.
5. **ALU**: Based on opcode, fcode and inputs, computes the flags and Output of ALU.
6. **Branch**: Decides the value of PCSrc (flag) and ExNPC from the opcode and fcode, hence adding the branching part to the processor.
7. **writeAddress**: Decides the values of Address of the register, data and the RegWrite flag which is used as input to write data into ALU.
8. **Data Memory**: Block RAM used to load and store data elements.

Assumptions and Other Points to be Noted:

1. Since it is not mentioned in the assignment the mode of addressing to be used, we have used direct addressing for branch instructions.
2. Block RAMs have a peculiar issue that they have significant delay in fetching data (around 0.5 clock cycles) from the RAM. Hence, we have divided the input clock into two parts, a slower one and a faster one. The faster one is eight times faster and is used to fetch data from Block RAM whereas the slower one is used for other modules.
3. Block RAMs have addresses as 0, 1, 2, Hence, we have used PC+1 instead of PC+4.
4. Due to the problem of Block RAMs having certain delay associated with fetch, in the case of branch instructions, there was a need of adding some pseudo-instructions that do nothing but add 0 to a random register so as to allow flow of control.

Running the Linear Search Algorithm:



The screenshot displays two side-by-side panels in a code editor. The left panel, titled 'AssemblyCode.asm', contains 26 lines of MIPS assembly code. The right panel, titled 'MachineCode.cpp', shows the corresponding 32-bit binary machine code for each assembly instruction, aligned line-by-line. The assembly code includes instructions for adding, comparing, loading, and branching, with labels 'LoopLabel', 'Match', and 'Exit' used for control flow. The machine code is represented as a sequence of 32-bit hexadecimal strings.

AssemblyCode.asm	MachineCode.cpp
0. addi \$t0 , 0	00100111000000000000000000000001,
1. add \$t1 , \$zero	000010000000000000000000110000000000,
2. addi \$s2 , 10	0011011000000000000000000000101001,
3. comp \$s3 , \$s2	0001011110110000000001000000000000,
4. addi \$s3 , 1	001101110000000000000000000000101,
5. lw \$s0 , 10(\$zero)	010101000000000000000000000010100,
6. comp \$s1 , \$s0	0001010110100000000001000000000000,
7. addi \$s1 , 1	001101010000000000000000000000101,
8. add \$s5 , \$zero	000110010000000000000000110000000000,
9. add \$t2 , \$t1	000010010100000000000000110000000000,
10. add \$t2 , \$s3	0000100110111000000001100000000000,
11. bz Exit	0110000000000000000000000110010001,
12. add \$s7 , \$zero	000110110000000000000000110000000000,
13. lw \$t3 , 0(\$t0)	0100101000111000000000000000000000,
14. add \$t4 , \$t3	0000101101010000000001100000000000,
15. add \$t4 , \$s1	0000100110101000000001100000000000,
16. bz Match	0110000000000000000000000101100001,
17. add \$s7 , \$zero	000110110000000000000000110000000000,
18. addi \$t1 , 1	001010000000000000000000000000101,
19. addi \$t0 , 4	001001110000000000000000000010001,
20. b LoopLabel	001000000000000000000000010010000,
21. add \$s7 , \$zero	000110110000000000000000110000000000,
22. addi \$s5 , 1	001110010000000000000000000000101,
23. b Exit	0110000000000000000000000110010000,
24. add \$s7 , \$zero	000110110000000000000000110000000000,
25.	1110000000000000000000000000000000;

The above assembly language code is written according to the assembly language specification provided to us.

The result of the Linear Search is stored in the register **\$s5**, which corresponds to the 26th register with an address of **25 (11001)** in our Memory.

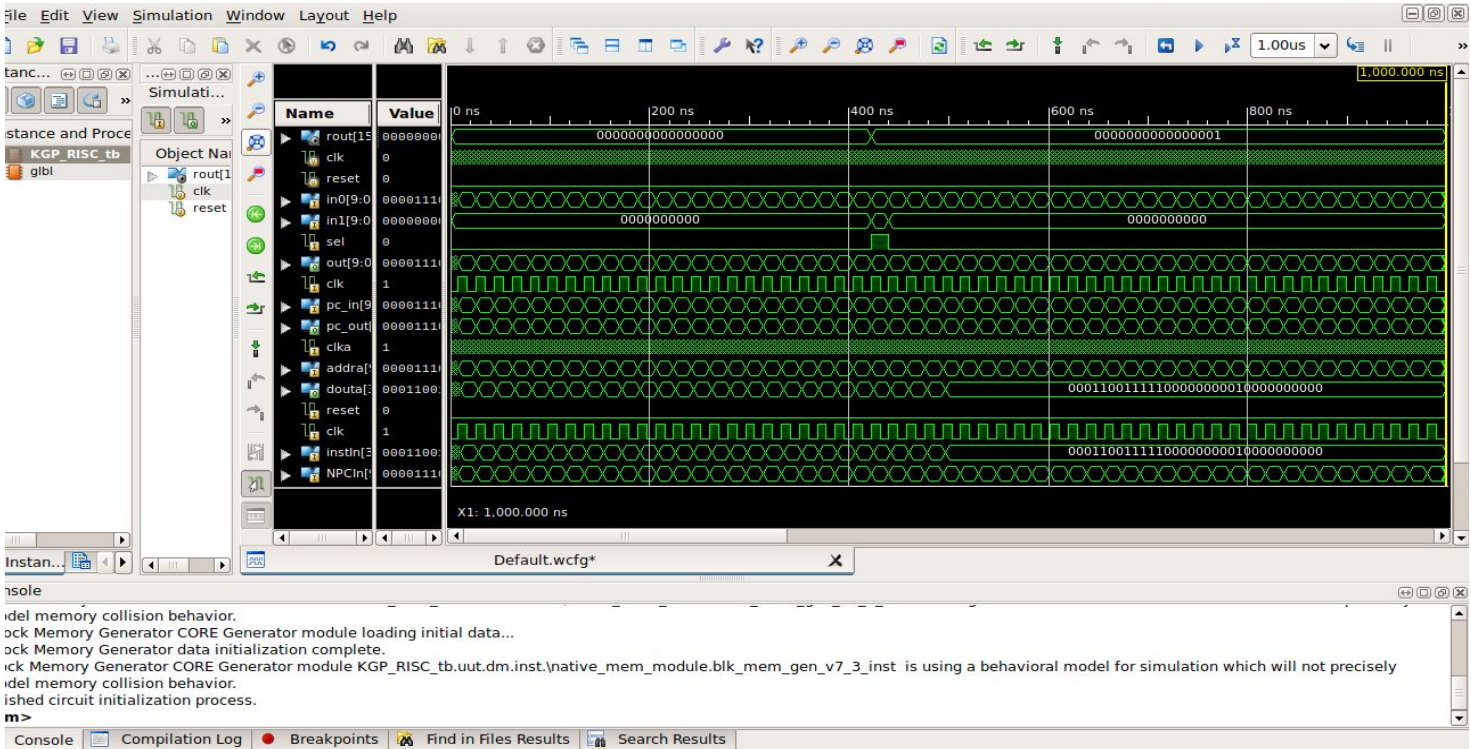
The Register holds a **Zero Value** in case there is **No Match** and a **Non-Zero** value in case there is a **Match**.

The corresponding Machine Language Instruction Sequence is generated according to the encoding of each instruction in our language specification, with appropriate values assigned to the opCode, rs, rt, shamt, imm, label, fCode depending on the instruction and the registers used.

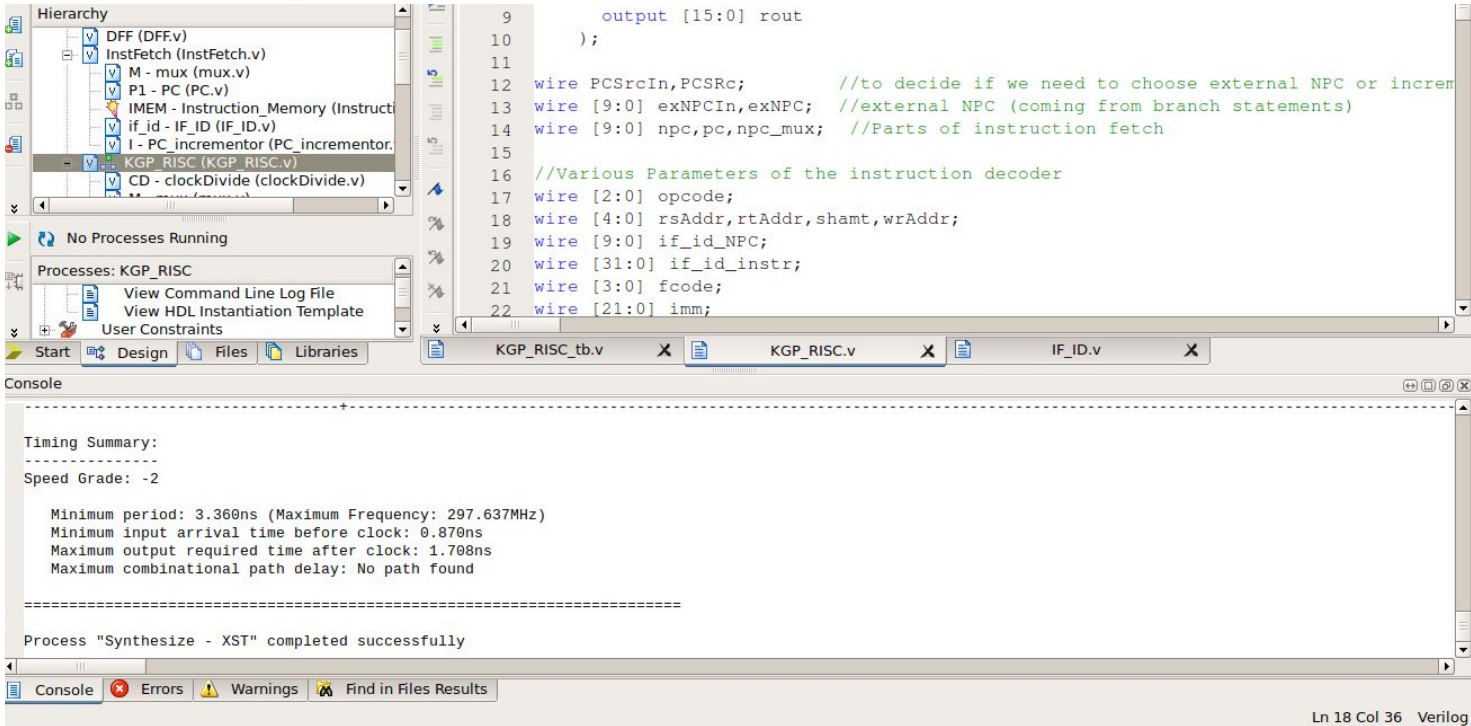
The last instruction with the first three bits set indicates an opCode of 7, which does not fall in any class of our specification, hence leading to the termination of the program.

In the above Screenshot, each Assembly Language Instruction (*left panel*) has its equivalent 32-bit Binary Representation in the Machine Language (*right panel*).

Simulation Screenshot:

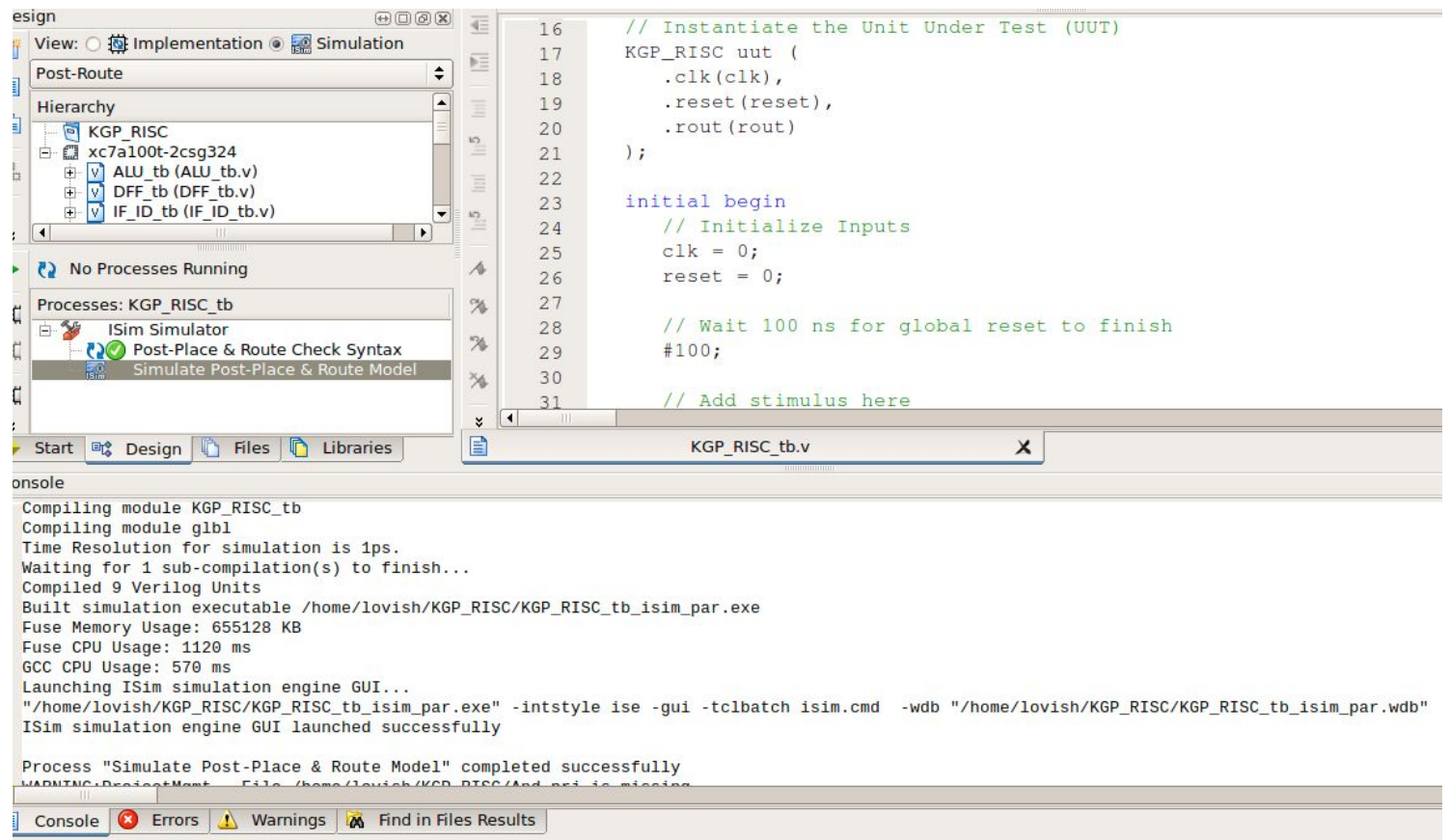


Behavioral Synthesis:



Post Route Simulation:

UCF File: Included with the ZIP File.



Hence this marks the **Successful completion of the Simulation of the KGP-RISC Processor.**