




KGP-RISC

Group No - 05

19CS30050 - Suryam Arnav Kalra

19CS30025 - Kunal Singh

Steps to Run the code :

1. With the file KGP_RISC_tb selected click on Simulate Behavioral Model.
2. On opening of new window click on the run all button  in the top menu, since by default in xilinx the program runs for 1 microsecond and we want to run the program till it completes.
3. To see the output, on the top left bar click on KGP_RISC_tb -> uut -> dm -> Always_29_0.
4. After this in the middle column you'll be able to see DMem[0:31,31:0] expand it and see the output (gcd) at [3,31:0] of DMem.
5. To change the operands for the gcd program you go to module DataMemory and change the operands at DMem[1] and DMem[2].

Register Convention :

The Register Usage convention for the available 32 registers is

Register Numbers	Register Name	Description
0-30	\$r0-r30	General purpose registers
31	\$ra	Function Return address

Instruction Set Architecture:

We have divided all the 22 instructions in two different types , first one is R-type Instruction which contains add, comp, and, xor, shll, shrl, shllv, shra, shrav and the second one is I-type Instructions which contains addi, compi, lw, sw, b, br, bltz, bz, bnz, bl, bcy, bncy.

1. R - Type Instructions:

a. Basic instruction format

opcode - 6	rs - 5	rt - 5	XXXXX	shamt - 5	func code - 6
------------	--------	--------	-------	-----------	---------------

b. Detailed instruction format

Operation	Usage	Meaning	opcode	R1	R2	Don't care	Shift amount	Function code
add	add rs,rt	$rs \leftarrow (rs) + (rt)$	000000	rs	rt	XXXXX	XXXXX	000000
comp	comp rs,rt	$rs \leftarrow 2's \text{ Complement } (rt)$	000000	rs	rt	XXXXX	XXXXX	000001
and	and rs,rt	$rs \leftarrow (rs) \wedge (rt)$	000000	rs	rt	XXXXX	XXXXX	000010
xor	xor rs,rt	$rs \leftarrow (rs) \oplus (rt)$	000000	rs	rt	XXXXX	XXXXX	000011
shll	shll rs, sh	$rs \leftarrow (rs)$ left-shifted by sh	000000	rs	XXXXX	XXXXX	sh	000100
shrl	shrl rs, sh	$rs \leftarrow (rs)$ right-shifte d by sh	000000	rs	XXXXX	XXXXX	sh	000101
shllv	shllv rs, rt	$rs \leftarrow (rs)$ left-shifted by (rt)	000000	rs	rt	XXXXX	XXXXX	000110
shrlv	shrlv rs, rt	$rs \leftarrow (rs)$ right-shifte d by (rt)	000000	rs	rt	XXXXX	XXXXX	000111
shra	shra	$rs \leftarrow (rs)$	000000	rs	XXXXX	XXXXX	sh	001000

	rs, sh	arithmetic right-shifte d by sh						
shrav	shrav rs, rt	$rs \leftarrow (rs)$ right-shifte d by (rt)	000000	rs	rt	XXXXX	XXXXX	001001

Here, XXXXX represents don't care values meaning they can be anything in the instruction, they won't affect its meaning.

Since we have an opcode = 000000 which is the same for every R-type instruction, and a function code field of 6 bits which is unique to every instruction, we could have a total of $2^6 = 64$ instructions out of which we have used 10 only.

Therefore, 54 more instructions could be added in R-type

2. I - Type Instructions:

a. Basic instruction format

opcode - 6	rs - 5	rt - 5	imm - 16
------------	--------	--------	----------

b. Detailed instruction format

Operation	Usage	Meaning	opcode	rs	rt	imm
addi	addi rs,imm	$rs \leftarrow (rs) + imm$	000001	rs	XXXX X	imm constant
compi	compi rs,imm	$rs \leftarrow 2's \text{ Complement } (imm)$	000010	rs	XXXX X	imm constant
lw	lw rt,imm(rs)	$rt \leftarrow mem[(rs) + imm]$	000011	rs	rt	imm constant

sw	sw rt,imm,(rs)	mem[(rs) + imm] ← (rt)	000100	rs	rt	imm constant
b	b L	goto L	000101	XXXXX	XXXX X	offset
br	br rs	goto (rs)	000110	rs	XXXX X	offset
bltz	bltz rs,L	if(rs) < 0 then goto L	000111	rs	XXXX X	offset
bz	bz rs,L	if (rs) = 0 then goto L	001000	rs	XXXX X	offset
bnz	bnz rs,L	if(rs) ≠ 0 then goto L	001001	rs	XXXX X	offset
bl	bl L	goto L; 31 ← (PC)+4	001010	XXXXX	XXXX X	offset
bcy	bcy L	goto L if Carry = 1	001011	XXXXX	XXXX X	offset
bncy	bncy L	goto L if Carry = 0	001100	XXXXX	XXXX X	offset

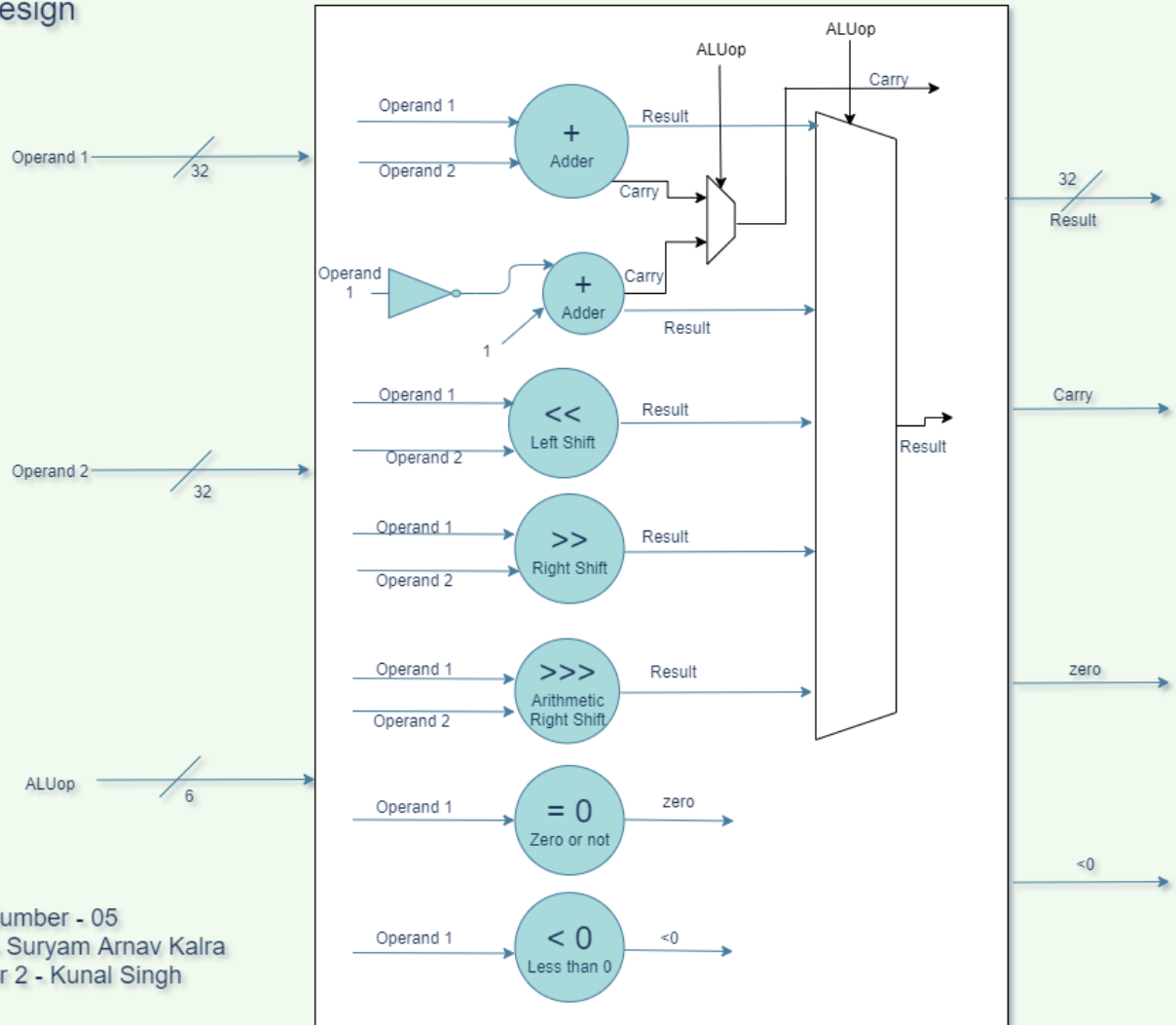
Here, XXXXX represents don't care values meaning they can be anything in the instruction, they won't affect its meaning.

Since we have an opcode field of 6 bits which is unique to every instruction, we could have a total of $2^6 = 64$ instructions out of which we have used 12 only and 1 more opcode is reserved for R-Type instructions.

Therefore, 51 more instructions could be added in I-type

Architecture of the processor (included in a separate pdf file)

ALU-Design



Truth Table

Flags from the control

	RegWrite	MemToReg	AluSrc	isShift	isLoadStore	MemRead	MemWrite	isJal	TAFR
add	1	0	0	0	0	0	0	0	0

comp	1	0	0	0	0	0	0	0	0
addi	1	0	1	0	0	0	0	0	0
compi	1	0	1	0	0	0	0	0	0
and	1	0	0	0	0	0	0	0	0
xor	1	0	0	0	0	0	0	0	0
shll	1	0	1	1	0	0	0	0	0
shrl	1	0	1	1	0	0	0	0	0
shllv	1	0	0	0	0	0	0	0	0
shrlv	1	0	0	0	0	0	0	0	0
shra	1	0	1	1	0	0	0	0	0
shrav	1	0	0	0	0	0	0	0	0
lw	1	1	1	0	1	1	0	0	0
sw	0	0	1	0	1	0	1	0	0
b	0	0	0	0	0	0	0	0	0
br	0	0	0	0	0	0	0	0	1
bltz	0	0	0	0	0	0	0	0	0
bz	0	0	0	0	0	0	0	0	0
bnz	0	0	0	0	0	0	0	0	0
bl	1	0	0	0	0	0	0	1	0
bcy	0	0	0	0	0	0	0	0	0
bncy	0	0	0	0	0	0	0	0	0

1. ALUop for different ALU operations

Operation	ALUop
Addition	000000
Complement	000001
AND	000010
XOR	000011
Logical left shift	000100
Logical right shift	000101
Arithmetic right shift	000110
Check for zero	000111
Check for less than zero	001000

2. Branchop for different Branch operations

Operation	Branchop
Branch on carry	000001
Branch on no carry	000010
Branch on flag zero	000011
Branch on flag not zero	000100
Branch on less than zero	000101
Unconditional Branch	000110
Branch on register	000110
Branch and link	000110

Here, unconditional branch, branch on register and branch & link have the same branchop as they have to just set the flag do_branch to 1 and rest of the work is done by the control.

Modules:

1. **ALU.v**: This module contains the main arithmetic and logical operations to be performed by the processor implementing addition, complementation, logical left and right shifts and arithmetic right shifts.
2. **Branch.v**: This module contains the main branching logic of the processor. It will branch to the desired label mentioned in the instruction on getting the branch instruction from the Control and necessary flags from the ALU.
3. **Control.v**: This is the heart of the processor setting various flags to 1 and 0 on seeing the instruction so as to properly execute the instruction.
4. **DataMemory.v**: This module contains the 32 bit data memory of the program in which we can store the values of the variables used by the code/processor during execution.
5. **KGP_RISC.v**: This is the main top module of the design which instantiates all the other modules and performs the desired operations on the tick of the clock.
6. **KGP_RISC_tb.v**: This is the test bench for the processor which initializes the clock and reset signals and starts the execution of the processor.
7. **InstructionMemory.v**: This module contains the set of 32-bit instructions which will be executed by the processor during its execution.
8. **RegisterFile.v**: This module contains the 32-bit 32 registers to be used by the processor during its execution. The register number 31 is kept for return addresses.
9. **PC.v**: This module contains the program counter and sets it to the next value after each tick of the clock.
10. **SignExtension.v**: This module is used to convert the 16-bit values to 32-bit values.

Flags/Controls:

1. **RegWrite**: Flag to decide whether something has to be written to the destination register or not.
2. **MemToReg**: Flag to decide whether to write data from memory or ALU result to the register.

3. **ALUSrc**: Flag to decide whether to give 16-bit (32-bit extended) value or the value stored in the register to the ALU as second operand.
4. **isShift**: Flag to decide whether it is a shift operation or not.
5. **isLoadStore**: Flag to decide whether it is a load/store (lw/sw) operation or not.
6. **MemRead**: Flag to decide whether data has to be read from memory or not.
7. **MemWrite**: Flag to decide whether data has to be written to memory or not.
8. **isJAL**: Flag to decide whether the instruction is Jump and Link (branch and link) or not.
9. **TAFR(take address from register)**: Flag to take the jump address from the register (useful in branch and link instructions).
10. **ALUOp**: 6-bit flag to control the operation of the ALU.
11. **Branchop**: 6-bit flag to control the operation of the branching module.
12. **do_branch(from Branch)**: Flag to denote whether branching has to be done or not.
13. **carry(from ALU)**: Flag from the ALU to denote if there is a carry after the operations on the operands.
14. **zero(from ALU)**: Flag from the ALU to denote if the result of the operation of the ALU is zero or not.
15. **lessThanZero(from ALU)**: Flag from the ALU to denote whether the result of the operation of the ALU is less than zero or not.

Source code for ComputeGCD:

```

IMem[0] = 32'b00001100000000000000000000000000; // lw r0 = 0
IMem[1] = 32'b00001100000000001000000000000001; // lw r1 DMem[1]
IMem[2] = 32'b00001100000000010000000000000010; // lw r2 DMem[2]
IMem[3] = 32'b0010000000100000000000000000010001; // bz r1 exit1
IMem[4] = 32'b0010000001000000000000000000010011; // bz r2 exit2
IMem[5] = 32'b000000000110001000000000000000001; // comp r3, r2
IMem[6] = 32'b000011000000001000000000000000000; // lw r4 = 0
IMem[7] = 32'b000000000100000010000000000000000; // add r4, r1
IMem[8] = 32'b000000000100000110000000000000000; // add r4, r3
IMem[9] = 32'b000111001000000000000000000001011; // bltz r4, L1 = 11
IMem[10] = 32'b000101000000000000000000000001110; // b L2 = 14
IMem[11] = 32'b000000000101000010000000000000001; // comp r5, r1
IMem[12] = 32'b000000000100010100000000000000000; // add r2, r5
IMem[13] = 32'b000101000000000000000000000000011; // b 3 (loop)

```

```

IMem[14] = 32'b0000000001010001000000000000000001; // comp r5, r2
IMem[15] = 32'b0000000000010010100000000000000000; // add r1, r5
IMem[16] = 32'b0001010000000000000000000000000011; // b 3 (loop)
IMem[17] = 32'b0001000000000010000000000000000011; // sw r2 DMem[3]
IMem[18] = 32'b0001010000000000000000000000010100; // b 20
IMem[19] = 32'b0001000000000000100000000000000011; // sw r1 DMem[3]

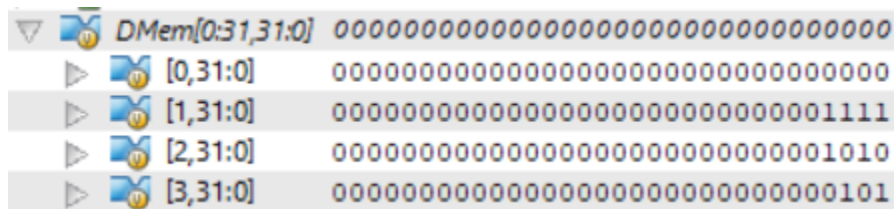
```

Sample input/output on execution of above code:

1. Execution of ComputeGCD(15, 10)

Input: DMem[1] = 15, DMem[2] = 10

Output: DMem[3] = 5 (gcd of 15 and 10)



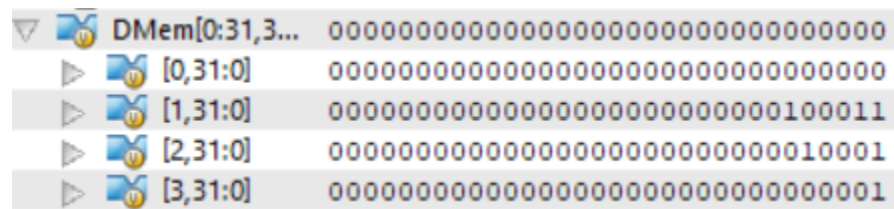
Index	Value (Hex)	Value (Binary)
DMem[0:31,31:0]	00000000000000000000000000000000	00000000000000000000000000000000
[0,31:0]	00000000000000000000000000000000	00000000000000000000000000000000
[1,31:0]	000000000000000000000000000000001111	0000000000000000000000000000001111
[2,31:0]	000000000000000000000000000000001010	0000000000000000000000000000001010
[3,31:0]	00000000000000000000000000000000101	000000000000000000000000000000101

Fig a) Data Memory

2. Execution of ComputeGCD(35, 17)

Input: DMem[1] = 35, DMem[2] = 17

Output: DMem[3] = 1 (gcd of 35 and 17)



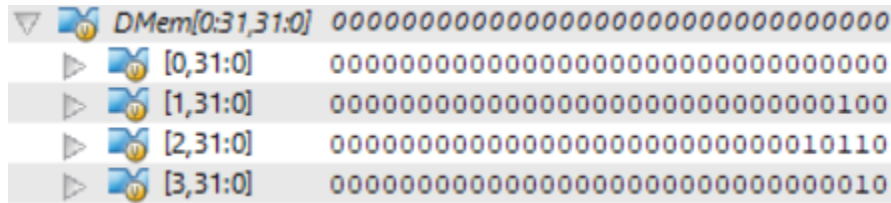
Index	Value (Hex)	Value (Binary)
DMem[0:31,31:0]	00000000000000000000000000000000	00000000000000000000000000000000
[0,31:0]	00000000000000000000000000000000	00000000000000000000000000000000
[1,31:0]	00000000000000000000000000000000100011	000000000000000000000000000000100011
[2,31:0]	0000000000000000000000000000000010001	00000000000000000000000000000010001
[3,31:0]	0000000000000000000000000000000000001	0000000000000000000000000000000000001

Fig b) Data Memory

3. Execution of ComputeGCD(4, 22)

Input: DMem[1] = 4, DMem[2] = 22

Output: DMem[3] = 2 (gcd of 4 and 22)



▼ DMem[0:31,31:0]	00000000000000000000000000000000
▶ [0,31:0]	00000000000000000000000000000000
▶ [1,31:0]	00000000000000000000000000000100
▶ [2,31:0]	000000000000000000000000000010110
▶ [3,31:0]	00000000000000000000000000000010

Fig c) Data Memory

Assumptions:

1. We have used **direct mode of addressing** for branch instructions.
2. We have hard-coded the instructions for the execution of the GCD computation in the instruction memory.
3. We have used PC+1 instead of PC+4.
4. The value of PC is initialized with 0.
5. **Note: Since the code takes more than 1 microsecond to run, please click the Run ALL button in the simulation window to completely execute the program.**