Computer Organisation and Architecture Laboratory KGP-miniRISC

Group 25
Rounak Saha - 20CS30043
Swarup Padhi - 20CS30062

1. Instruction encoding:

Each instruction is of 32 bits and following are the different instruction types:

1. R_FORMAT:

12 instructions = {add, comp, diff, and, xor, shll, shrl, shllv, shra, shrav, shrlv}

opcode (6 bits)	rs(5 bits)	rt(5 bits)	don't care (5 bits) (xxxxx)	shamt(5 bits)	func(6 bits)

2. I_FORMAT(FIRST TYPE)

2 instructions = {addi, compi}

opcode(6 bits)	rs(5 bits)	immediate value(21 bits)

3. I_FORMAT(SECOND TYPE)
2 instructions = {lw, sw}

opcode (6 bits)	rs(5 bits)	rt(5 bits)	immediate address (16 bits)

exact address stored in the last 16 bits of immediate address.

4. BRANCH_FORMAT

3 instructions = {bz, bltz, bnz}

Branching is conditional.

opcode (6 bits)	rs(5 bits)	branch offset(16 bits)

PC relative addressing mode used.
For obtaining the actual branch address (or label),
suppose “branch offset” is the last 16 bits of this instruction
branch address = PC + 1 + branch offset

5. **JUMP_FORMAT(FIRST TYPE)**

4 instructions = {b, bl, bcy, bncy}

b and bl are unconditional jumps.
bcy and bncy are conditional jumps.

opcode (6 bits)	jump address (26 bits)

Pseudo direct addressing mode used.
For obtaining the actual jump address (or label),
suppose “jump” is the last 26 bits of the instruction
jump address = {PC[31:26],jump}
concatenate the first 6 bits of PC to the left of “jump”\

6. **JUMP_FORMAT(SECOND TYPE)**

1 instruction = {br}

br is an unconditional jump.

opcode (6 bits)	rs(5 bits)	don’t care(15 bits)	func(6 bits)

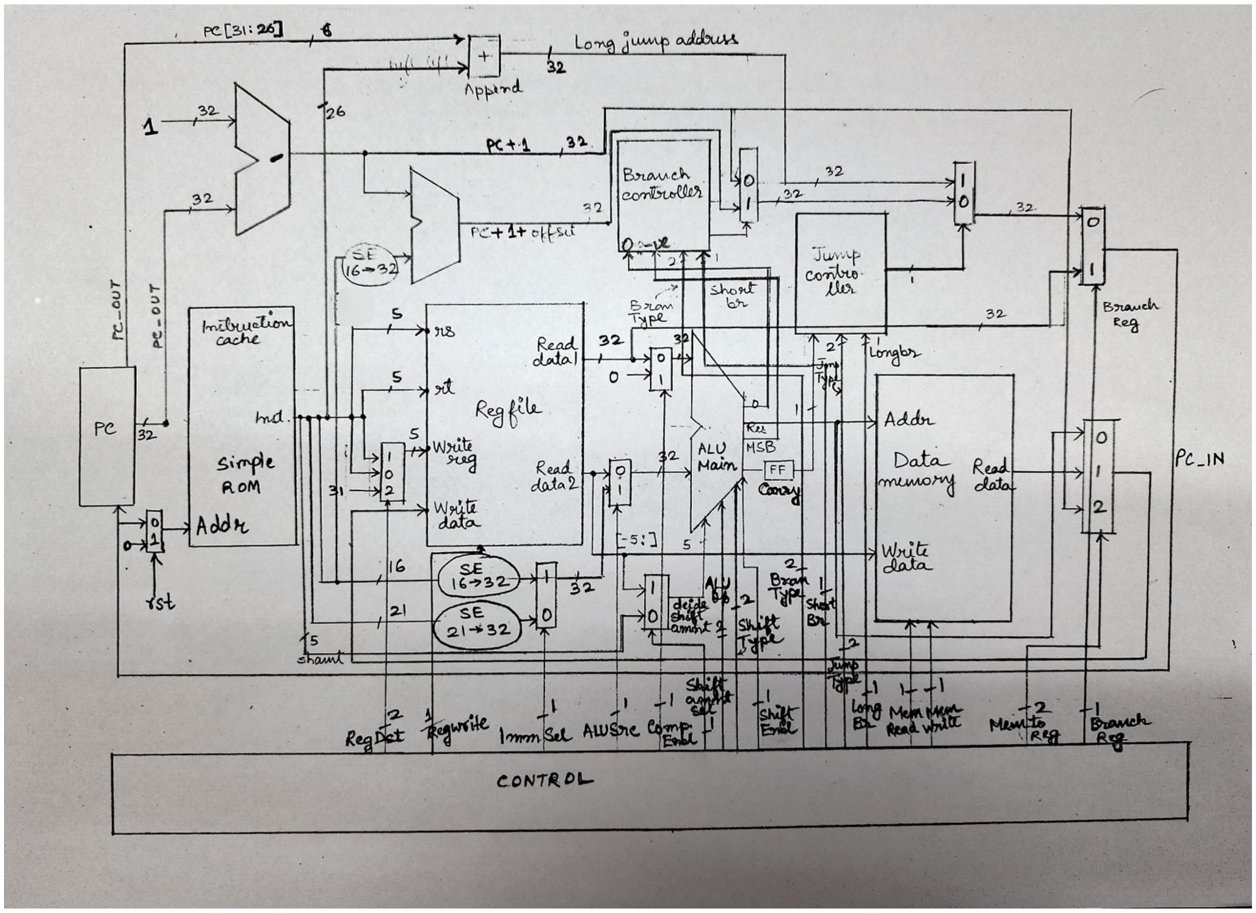
Direct addressing mode used.
rs stores the exact address of the instruction to which it has to jump.

The different instructions are encoded in the following format:

		OP[2:0]							
		000	001	010	011	100	101	110	111
OP [5:3]	000	R-fmt							
	001	addi	compi						
	010								
	011		lw						
	100	b	bl					bcy	bncy
	101				bltz		bz	bnz	
	110								
	111				sw				

		func[2:0]							
		000	001	010	011	100	101	110	111
func [5:3]	000	shll	shrl	shra		shllv	shrlv	shrav	
	001	add	comp						
	010	xor							
	011	and							
	100	diff							
	101								
	110								
	111								br

2. Datapath design



3. Control signals description

1. **RegDst =** Control signal to 3:1 mux selecting the register to which data is to be written between rt,rs and register numbered 31(stored the address of last instruction for bl type instruction)

2. **ImmSel =** Control signal to 2:1 mux for selecting the immediate value bits(between size 21 and 16 bits)to feed to the ALU.

3. **ALUSrc =** Control signal to 2:1 mux for selecting the second source to the ALU between immediate value and rt.

4. **AluOP(2 bits) =** Control signal to 4:1 mux for selecting between the ALU operations and,xor,add and diff.

5. **Comp Enbl. =** Control signal to 2:1 mux for selecting between 0 and rs to pass as input to the first source of ALU (used for compliment type instructions)

6. **Shift Amnt. =** Control signal to 2:1 mux for selecting the shifting value between rs and shamt.

7. **Shift Enbl. =** Control signal to shifter module to enable performing shifting operations on input.

8. **Shift Type(2 bits) =** Control signal to 3:1 mux for selecting between the type of shifting to be done (shift left logical,shift right logical and shift right arithmetic)

9. **Branch Type(2 bits) =** Control signal to provide the branch type being asked by the instruction between blitz,bz and bnz.

10. **ShortBr =** Control signal to specify if branching is via an offset provided in instructions bltz,bz and bnz.

11. **Jump Type(2 bits) =** Control signal to provide the branch type being asked by the instruction between b,bl,bcy and bncy,

12. **Longbr =** Control signal to specify if branching done by b,bl,bcy or bncy.

13. **MemRead =** Control signal to data memory ram to enable reading from memory.

14. **MemWrite =** Control signal to data memory ram to enable writing to memory.

15. **MemToReg(2 bits) =** Control signal to 3:1 mux for choosing between the outputs of ALU,the data memory and the PC value to pass as write value for register.

16. **Branch Reg =** Control signal to 2:1 mux to select the between the PC from long branch and short branch and the register address to write onto PC.

17. **RegWrite =** Control signal to reg file to enable writing onto registers in it.

4. Truth table for control signals

instruction	OPCODE	FUNC	Reg Dst	Imm Sel	ALU Src	ALU Op	Comp Enbl.	Shift Amnt	Shift Enbl	Shift Type	Branch Type	ShortBr	Jump Type	Long Br	Mem Read	Mem Write	Mem ToReg	Branch Reg	Reg Write
add rs,rt	000000	001000	00	x	0	01	0	x	0	x	x	0	xx	0	0	0	00	0	1
comp rs,rt	000000	001001	00	x	0	01	1	x	0	x	x	0	xx	0	0	0	00	0	1
addi rs,imm	001000	NA	00	0	1	01	0	x	0	x	x	0	xx	0	0	0	00	0	1
compi rs,imm	001001	NA	00	0	1	01	1	x	0	x	x	0	xx	0	0	0	00	0	1
lw,rt,imm,rs	011001	NA	01	1	1	01	0	x	0	x	x	0	xx	0	1	0	01	0	1
sw rt,imm,rs	111011	NA	xx	1	1	01	0	x	0	x	x	0	xx	0	0	1	xx	0	0
and rs,rt	000000	011000	00	x	0	11	0	x	0	x	x	0	xx	0	0	0	00	0	1
xor rs,rt	000000	010000	00	x	0	10	0	x	0	x	x	0	xx	0	0	0	00	0	1
shll rs,sh	000000	000000	00	x	x	x	0	0	1	00	x	0	xx	0	0	0	00	0	1
shrl rs,sh	000000	000001	00	x	x	x	0	0	1	01	x	0	xx	0	0	0	00	0	1
shllv rs,rt	000000	000100	00	x	x	x	0	1	1	00	x	0	xx	0	0	0	00	0	1
shrlv rs,rt	000000	000101	00	x	x	x	0	1	1	01	x	0	xx	0	0	0	00	0	1
shra rs,sh	000000	000010	00	x	x	x	0	0	1	10	x	0	xx	0	0	0	00	0	1
shrav rs,rt	000000	000110	00	x	x	x	0	1	1	10	x	0	xx	0	0	0	00	0	1
b L	100000	NA	xx	x	x	x	x	x	0	x	x	x	00	1	0	0	xx	0	0
br rs	000000	111111	xx	x	x	x	x	x	0	x	x	x	xx	x	0	0	xx	1	0
bltz rs,L	101011	NA	xx	x	x	x	0	x	0	x	01	1	xx	0	0	0	xx	0	0
bz rs,L	101101	NA	xx	x	x	x	0	x	0	x	10	1	xx	0	0	0	xx	0	0
bnz rs,L	101110	NA	xx	x	x	x	0	x	0	x	11	1	xx	0	0	0	xx	0	0

bl L	100001	NA	10	x	x	x	x	x	0	x	x	x	01	1	0	0	10	0	1
bcy L	100110	NA	xx	x	x	x	x	x	0	x	x	0	10	1	0	0	xx	0	0
bncy L	100111	NA	xx	x	x	x	x	x	0	x	x	0	11	1	0	0	xx	0	0
diff rs,rt	000000	100000	00	x	0	00	0	x	0	x	x	0	xx	0	0	0	00	0	1

5. Datapath elements

- **Instruction memory:**

Single port ROM with Width(word length) = 32 and Depth(total capacity in terms of number of words or instructions) = 4096

Input:
Addr: address instruction to be fetched in the current cycle
clk

Output: 32-bit instruction, arrives at the next posedge after the address line is applied

(
Special note:
Since there the instruction is fetched at the next posedge after applying the address line, we feed the address of the next instruction to be executed directly into the instruction address port of the instruction memory instead of passing it through the PC, the next address is fed to the PC as well. Hence in the next posedge PC is updated and the instruction corresponding to that is available in the instruction line)

- **Register file:**

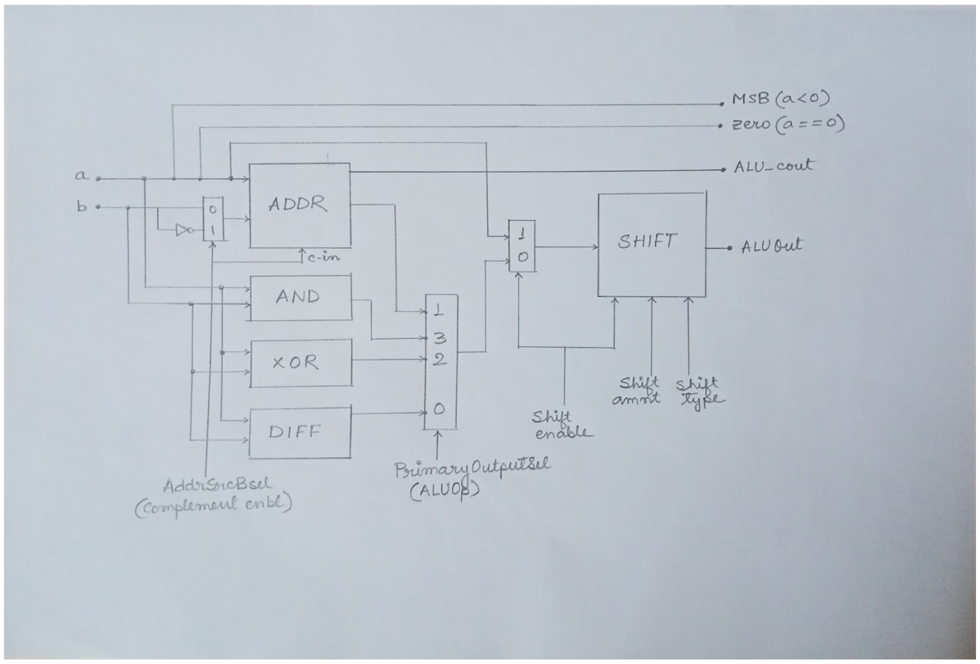
Consists of 32 32-bit registers.

Input:
rs, rt: two 5-bit input address ports
WriteReg: 5-bit address of register to write to if RegWrite is set
RegWrite: write enable line for regfile
WriteData: 32-bit data to be written to the register addressed WriteReg line id RegWrite is set
clk

Output:
ReadData1: 32-bit output of register addressed by rs

ReadData2: 32-bit output of register addressed by rt

- ## Arithmetic Logic Unit (ALU)



Inputs:

a, b: 32-bit operands
AddrSrcBSel: 1-bit, in case of complement type instructions, the second input to the adder should be bitwise complement of b which is decided by AddrSrcBSel (connected to CompEnbl control signal in the top module), the same bit is used to feed the carry-in the adder, so that the result is 2's complement of b (in case of complement type instructions input a is set 0 from the top module)
PrimaryOutputSel: 2-bit, this is basically the ALUOp which decides whether the desired ALU operation is ADD/AND/XOR/DIFF
ShiftEnbl: 1-bit line which is set if the instruction is of shift type
ShiftType: 2-bit input to decide type of shift (left logical/right logical/left arithmetic/right arithmetic), useful if and only if ShiftEnbl is set
ShiftAmnt: 5-bit value indicating the amount of shift(if required), hence useful only if ShiftEnbl is set

(Special note on the shifter module:

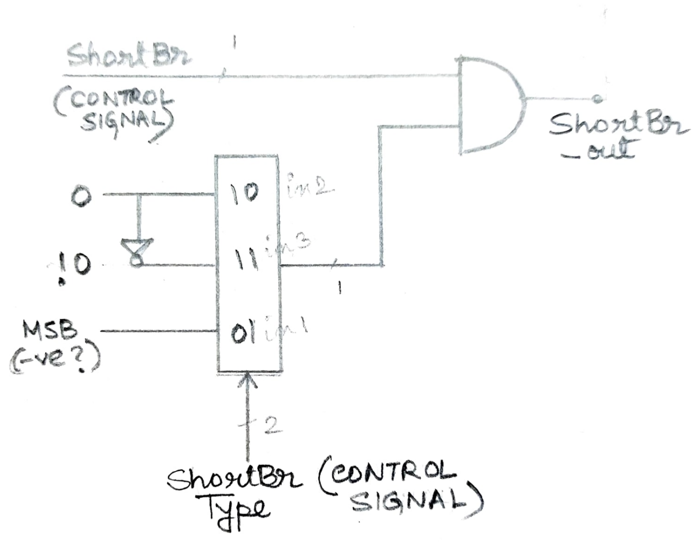
All outputs through the ALU are passed through the shifter module, although for our instruction set this is of no significance.
A 2_to_1 MUX decides the input to the shifter module should be the first operand or the result of the primary operations like ADD/AND/XOR/DIFF.
The selector line of this mux is the shift enable line itself since in all cases of shift, the shift is performed on rs only, this feature is specific to our ISA and might need to change if the ISA is to be extended to support other instructions of the form:

addsh rs,rt,imm   [rs <- (rs+rt)<<imm n]

This is in fact the main purpose of passing all kinds of outputs of the ALU through the shifter module, in our ISA since the shift instructions are specific and are not embedded as as a sub-operation in any of the other instructions, the shift enable line is set only in the shift operations)

Outputs:
ALUOut: 32-bit result of ALU operation
ALU_Cout: 1-bit carry-out of ALU arithmetic operation
MSB: 1-bit output indicating if the first operand is <0
zero: 1-bit output indication if the first operand is 0

- **Branch Decider Module**

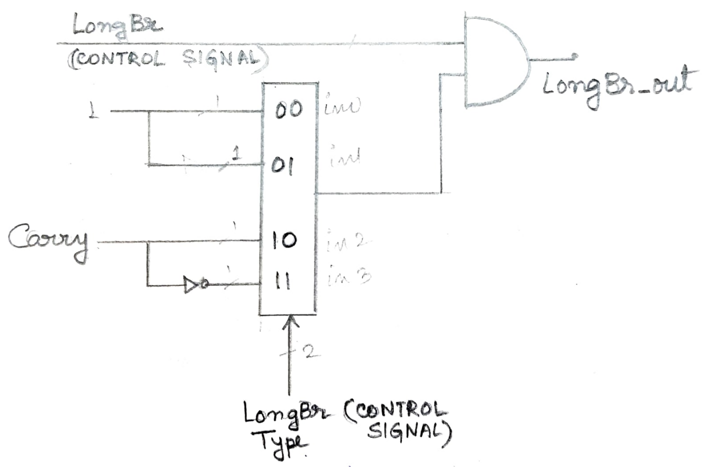


BRANCH DECIDER

Inputs:
zero: whether rs is 0 (received from ALU)
LongBr: whether the current instruction is a branch instruction
ShortBrType: 2-bit input to decide condition of branch (bz/bnz/bltz)

Output:
ShortBr_out: 1 bit output of final decision whether a short branch is to be taken, will be fed to a MUX which selects between PC+1 and PC+1+offset lines

- **Jump Decider module**



JUMP DECIDER

Inputs:
carry: whether the carry flag is set (received from the carry flip flop which stores the carry bit of the previous arithmetic operation)
LongBr: whether the current instruction is a jump instruction
LongBrType: 2-bit input to decide type of branch (b/bl/bcy/bncy)

Output:
LongBr_out: 1 bit output of final decision whether a long branch is to be taken, will be fed to a MUX which selects between immediate branch address and output of the MUX which takes the branch decision.

- ## Data Memory:

Single port RAM with Width = 32 and depth = 1024

Input:
Addr: memory address to be read/written to
WriteEnbl: 1-bit line to enable writing to memory
WriteData: Data to be written to if WriteEnbl is set
ReadData: value stored in memory location provided by Addr port

- ## Others:

PC Register: stores the address of the current instruction being executed

Carry Register: stores the carry bit of the previous arithmetic instruction

Other **MUXs** and **ADDER**s driven by control signals for next instruction calculation, branching decisions and selecting module inputs.

6. Testing on Bubble Sort:

(please refer to `CPU_FPGA_TB.v`)

1. **Bubble sort:**

The assembly code for bubble sort is in `mod_bubble_sort.s` file, we use the assembler to generate `mod_bubble_sort.coe` file which is loaded to the instruction memory. Following are the details of a sample run. The data memory is loaded with an array of 10 unsorted integers with base address 0 as follows:

Filename: `bubble_sort_mem.coe`

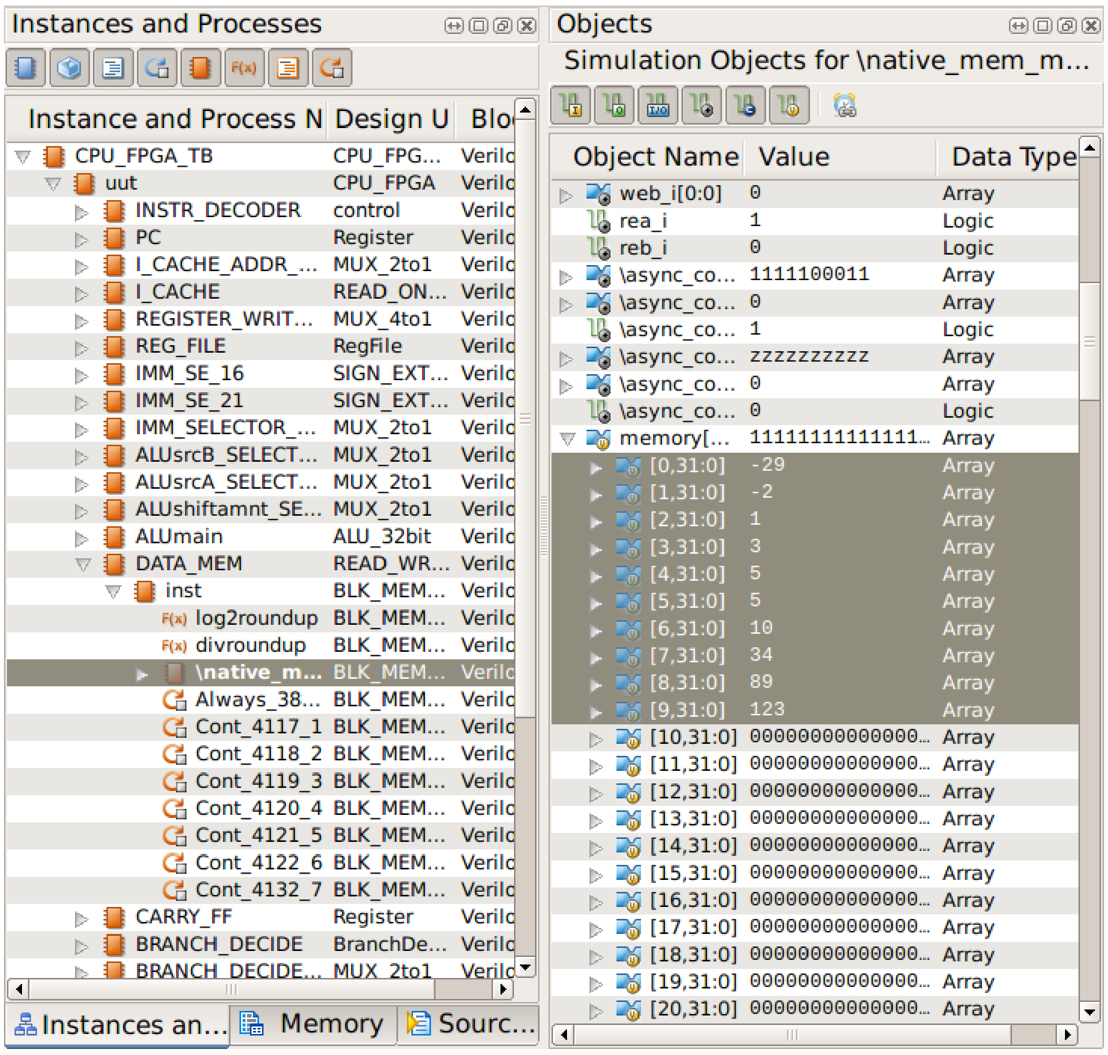
```
memory_initialization_radix=10;
memory_initialization_vector=
10,
-2,
5,
-29,
34,
123,
3,
1,
5,
89,
0
```

(Note that the terminal 0 is not a part of the 10 element array we intend to sort)

While demonstration, it is difficult for us to show that the array stored in data memory is actually unsorted before the execution starts, since we cannot directly fetch values from the BRAM using input signals (switches) in FPGA, also rst is 1 hence none of the registers are getting written to and output from them is 0. It can be verified from the .coe file for the bubble sort data memory that the array is actually unsorted

We let the instructions for bubble sort get executed from the instruction memory and after a sufficient number of clock cycles when the main sorting operation is complete load MEM[0], MEM[1], …, MEM[9] into the registers $0,$1,...,$9.

The $0 through $9 registers are then viewed one by one by setting output_sel in an appropriate manner to verify the correctness of the sorting operation. Following are the results of simulation:



Instances and Processes

Instance and Process N	Design U	Blo
CPU_FPGA_TB	CPU_FPG...	Verilo
uut	CPU_FPGA	Verilo
INSTR_DECODER	control	Verilo
PC	Register	Verilo
I_CACHE_ADDR_...	MUX_2to1	Verilo
I_CACHE	READ_ON...	Verilo
REGISTER_WRIT...	MUX_4to1	Verilo
REG_FILE	RegFile	Verilo
IMM_SE_16	SIGN_EXT...	Verilo
IMM_SE_21	SIGN_EXT...	Verilo
IMM_SELECTOR_...	MUX_2to1	Verilo
ALUsrcB_SELECT...	MUX_2to1	Verilo
ALUsrcA_SELECT...	MUX_2to1	Verilo
ALUshiftamnt_SE...	MUX_2to1	Verilo
ALUmain	ALU_32bit	Verilo
DATA_MEM	READ_WR...	Verilo
inst	BLK_MEM...	Verilo
log2roundup	BLK_MEM...	Verilo
divroundup	BLK_MEM...	Verilo
\native_m...	BLK_MEM...	Verilo
Always_38...	BLK_MEM...	Verilo
Cont_4117_1	BLK_MEM...	Verilo
Cont_4118_2	BLK_MEM...	Verilo
Cont_4119_3	BLK_MEM...	Verilo
Cont_4120_4	BLK_MEM...	Verilo
Cont_4121_5	BLK_MEM...	Verilo
Cont_4122_6	BLK_MEM...	Verilo
Cont_4132_7	BLK_MEM...	Verilo
CARRY_FF	Register	Verilo
BRANCH_DECIDE	BranchDe...	Verilo
BRANCH_DECIDE...	MUX_2to1	Verilo

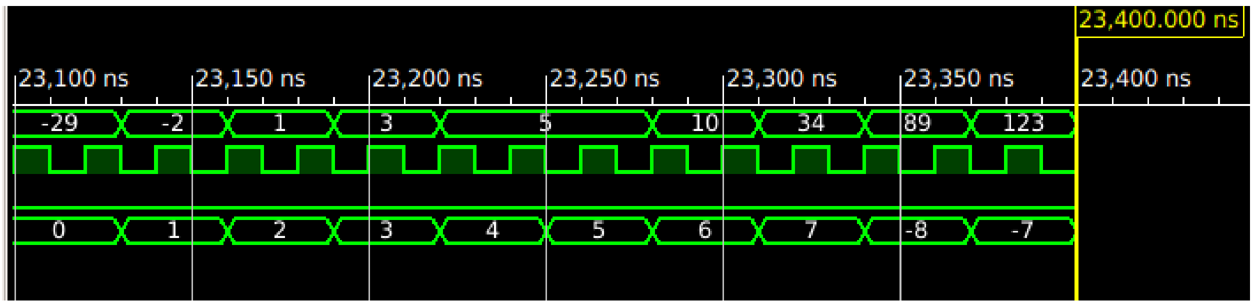
Instances an... Memory Sourc...

Objects

Simulation Objects for \native_mem_m...

Object Name	Value	Data Type
web_i[0:0]	0	Array
rea_i	1	Logic
reb_i	0	Logic
\async_co...	1111100011	Array
\async_co...	0	Array
\async_co...	1	Logic
\async_co...	zzzzzzzzzz	Array
\async_co...	0	Array
\async_co...	0	Logic
memory[...	11111111111111...	Array
[0,31:0]	-29	Array
[1,31:0]	-2	Array
[2,31:0]	1	Array
[3,31:0]	3	Array
[4,31:0]	5	Array
[5,31:0]	5	Array
[6,31:0]	10	Array
[7,31:0]	34	Array
[8,31:0]	89	Array
[9,31:0]	123	Array
[10,31:0]	00000000000000...	Array
[11,31:0]	00000000000000...	Array
[12,31:0]	00000000000000...	Array
[13,31:0]	00000000000000...	Array
[14,31:0]	00000000000000...	Array
[15,31:0]	00000000000000...	Array
[16,31:0]	00000000000000...	Array
[17,31:0]	00000000000000...	Array
[18,31:0]	00000000000000...	Array
[19,31:0]	00000000000000...	Array
[20,31:0]	00000000000000...	Array

Contents of data memory after sorting operation is complete



We set output_sel signal(lower most signal in the image) from 0 to 9 in order and the contents for the corresponding register are obtained in the output(uppermost signal in the image)