CS39001 : COMPUTER ORGANISATION LABORATORY

Design of a KGP-miniRISC Processor

Group Number: **10**

Members:

Subham Ghosh (20CS10065)
Anubhav Dhar (20CS30004)

Problem Statement

Our processor KGP-miniRISC has the following Instruction Set Architecture (ISA). Assume that the processor has a 32 bit word, with all the registers and memory elements having 32 bit data. The address line of the memory is also 32 bits.

Class	Instruction	Usage	Meaning
Arithmetic	Add	`add rs,rt`	$rs \leftarrow (rs) + (rt)$
	Complement	`comp rs,rt`	$rs \leftarrow$ 2's Complement (rs)
	Add Immediate	`addi rs,imm`	$rs \leftarrow (rs) + imm$
	Complement Immediate	`compi rs,imm`	$rs \leftarrow$ 2'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`	$rs \leftarrow rs$ left-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`	$rs \leftarrow$ 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 Less Than 0	`bltz rs,L`	if $(rs) < 0$ then goto L
	Branch On Flag Zero	`bz rs,L`	if $(rs) = 0$ then goto L
	Branch On Flag Not Zero	`bnz rs,L`	if $(rs) \neq 0$ then goto L
	Branch And Link	`bl L`	goto L; $31 \leftarrow (PC) + 4$
	Branch On Carry	`bcy L`	goto L if Carry = 1
	Branch On No Carry	`bncy L`	goto L if Carry = 0
Complex	Diff	`diff rs, rt`	$rs \leftarrow$ the LSB bit at which $rs$and rt differ

We are to develop first the op-code format for the above instruction set, identify the data path element and design the data path along with the control signals. Subsequently, we shall develop a single-cycle instruction execution unit for KGP-miniRISC.

Solution:

Instruction Encoding and Function Format:

The instruction encoding is attached in the table given below. The columns numbered from 0 to 31 represent the bits of the instruction in little-endian notation that is from the least significant bit to the most significant bit. For better clarity the decimal value of the function code (IR[5:0]) of each instruction is also present in the last column.

Some important points to note are:

- The functional code in the encoding of each instruction has a fixed width of 6 bits.
- The immediate operand in the `addi` and `compi` instruction is of 21 bits.
- Memory address offsets in `lw` and `sw` instruction has a width of 16 bits.
- All branch labels used in the ISA follow PC-relative addressing.
- `rs` and `rt` are the register operands and their addresses are encoded in 5 bits.
- The `sh` operand in shift instructions is encoded in 5 bits.
- The branch and link instruction stores the return address in the register numbered 31.

The instructions are encoded keeping in mind the future possibility of extending the ISA and maximum flexibility to a programmer. The functional codes are developed in such a way that control-level logic has the least combinational path delay.

Instruction Encoding Format

INS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	IR[5:0]
add	0	0	1	0	0	0	rs					rt																					4
comp	0	1	0	0	0	0	rs					rt																					2
addi	0	0	1	0	0	1	rs					imm																					36
compi	0	1	0	0	0	1	rs					imm																					34
nop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
diff	0	0	0	1	0	0	rs					rt																					8
and	0	1	1	0	0	0	rs					rt																					6
xor	1	0	0	0	0	0	rs					rt																					1
shll	1	0	1	0	0	1	rs					sh																					37
shrl	1	1	0	0	0	1	rs					sh																					35
shra	1	1	1	0	0	1	rs					sh																					39
shllv	1	0	1	0	0	0	rs					rt																					5
shrlv	1	1	0	0	0	0	rs					rt																					3
shrav	1	1	1	0	0	0	rs					rt																					7
lw	0	0	1	1	0	1	rt					rs					imm																44
sw	0	0	1	1	0	0	rt					rs					imm																12
b	0	0	0	0	1	1											L																48
br	0	0	0	0	1	0	rs																										16
bltz	1	0	0	0	1	1	rs										L																49
bz	0	1	1	0	1	1	rs										L																54
bnz	1	1	1	0	1	1	rs										L																55
bl	0	1	0	0	1	1	1	1	1	1	1						L																50
bcy	1	0	1	0	1	1											L																53
bncy	1	1	0	0	1	1											L																51
	FUNCTION CODE						RSA					RTA					OFFSET																

Datapath and Control Elements:

The design specifications of the datapath and control elements of our RISC processor are described below:

- A_MUX

  This is a 2×1 multiplexer that takes in input the read lines RSR and RTR and produces the input to the port A of the ALU depending on the 1-bit control signal SA. The output and input lines are of width 32-bits.

- B_MUX

  This is a 4×1 multiplexer that takes in input the read line RTR, and the lines RSR, IMM, OFF that comes from the encoding of the instruction stored in the register IR and produces the input to the port B of the ALU depending on the 2-bit control signal SB. The output and input lines are of width 32-bits. Sign extension is done for the lines IMM and OFF.

- ALU

  This is the arithmetic logic unit of the processor. It performs all the logical and arithmetic computations and produces the result through the line RES. The operation performed by the ALU depends upon the 3-bit control signal OP and the 1-bit control signal DIFF (which is used for the `diff` instruction). For addition operation the carry-out is stored in the 1-bit register CY. Note that the flags NZ, LZ, EZ and $\overline{\text{CY}}$ are not actually part of the ALU and are not stored. They are computed from RSR and CY using combinational logic directly to maximize efficiency and reduce delay.

- PC

  This is the program counter that contains the address of the instruction to be executed. It is a 32-bit register that is written with the address of the next instruction to be executed on the negative edge of the clock. The address of the next instruction to be executed comes from the line PC_IN. The value of the register comes out from the line PC_OUT.

- ADDER_L

  This is a 32-bit adder that performs the addition of PC_OUT with the label offset L (which is sign extended to 32-bits). The result comes out as the line PC_L.

- ADDER_1

  This is a 32-bit adder that increments the program counter by 1 unit. The result comes out as the line PC_1.

- PC_ADD_MUX

  This is a 2×1 multiplexer that takes in as input the 32-bit lines PC_L and PC_1 and produces the output through the 32-bit line PC_ADD depending on the 1-bit select line SAD.

- PC_MUX

  This is a 2×1 multiplexer that takes in as input the 32-bit lines PC_ADD and RSR and produces the output through the 32-bit line PC_IN depending on the 1-bit control signal SPC.

- REGISTER FILE

  The register file consists an array of 32, 32-bit general purpose registers that can be used by the programmer. There are two 5-bit address lines RTA and RSA that determine which registers to read and hence determine the value that comes out as 32-bit lines RSR and RTR. The 1-bit control line WR determines whether to write the value coming from the 32-bit line RSW to the register at address RSA on the negative edge of clock or not.

- RSW_MUX

  This is a 2×1 multiplexer that takes in as input the 32-bit lines RES and S and determines the output through the line RSW depending upon the 1-bit control signal SSW.

- S_MUX

  This is a 2×1 multiplexer that takes in as input two 32-bit lines, DOUTB and PC_1 and determines the output through the line S depending upon the 1-bit control signal SS.

- IR

  This stands for the 32-bit instruction that comes from the port DOUTA of the memory. Note that this is not a register. Since the output DOUTA from the RAM changes only at the positive edges of the clock, and since the processor follows a single-cycle execution cycle, so the instruction is not explicitly stored in a register as it remains constant during 1 clock cycle.

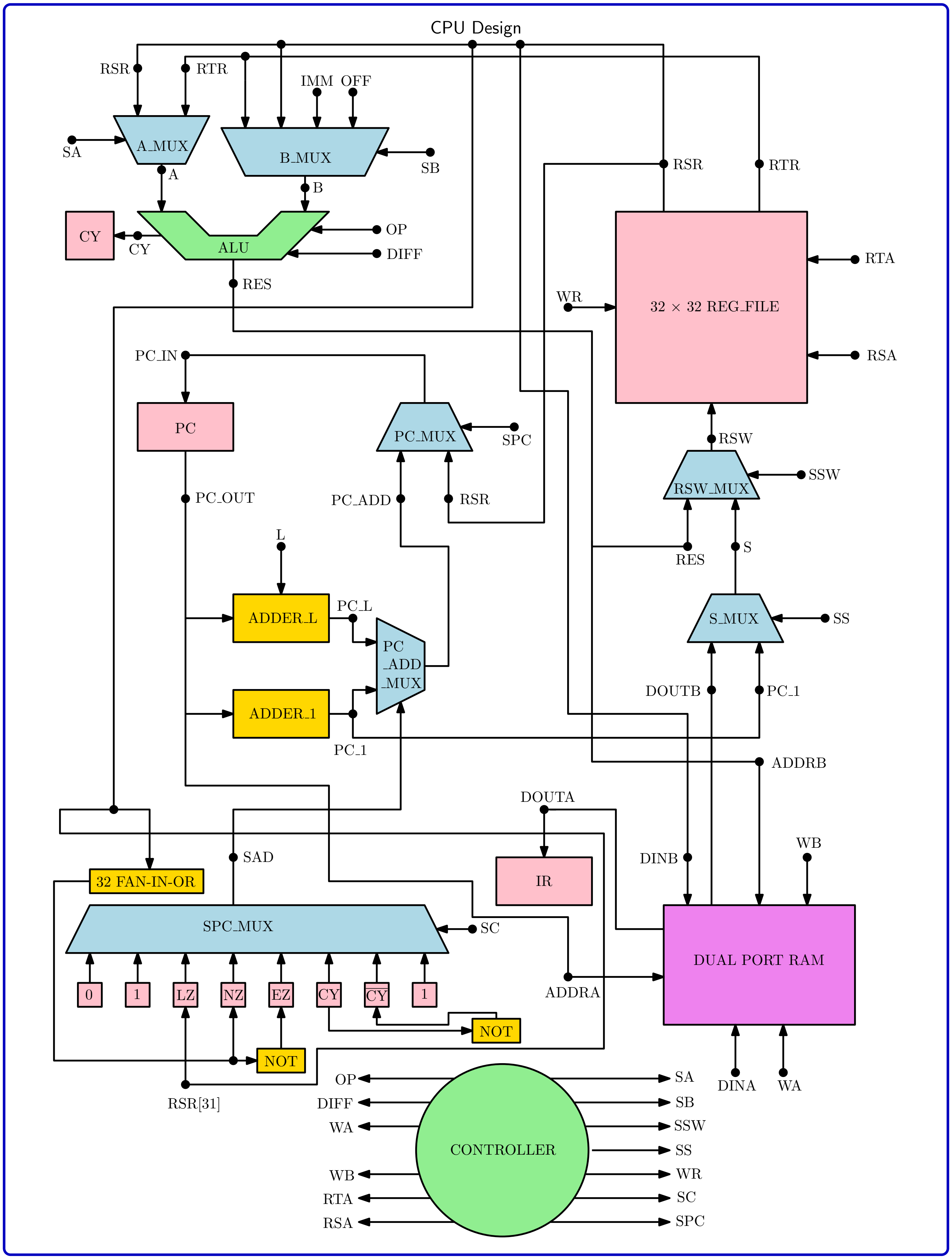
- SPC_MUX

  This is a 8×1 multiplexer that takes in as input 3 constant 1-bit lines, two of which are set to high (1) and the third is set to low (0). The remaining five 1-bit lines come from the flags LZ, NZ, EZ, CY, $\overline{\text{CY}}$. The output of this multiplexer determines what goes into PC_ADD_MUX as a select line during conditional and unconditional branches.

- FLAGS

  The flags are used during conditional branch instructions. The flags used by the processor are:

CY	This stands for the carry-flag which is 1 if there was an arithmetic overflow in the last addition operation.
NZ	This stands for the not-equal-to-zero flag which is 1 if RSR is non-zero. A 32-bit fan-in or is used to compute it.
EZ	This stands for the equal-to-zero flag which is 1 if RSR is equal to zero. It is computed by taking a not of NZ.
LZ	This stands for the less-than-zero flag which is 1 if RSR is less than zero. The most significant bit of RSR is used to compute it.
$\overline{\text{CY}}$	This stands for the not-carry-flag which is 1 if there was no arithmetic overflow in the last addition operation. It is computed by taking the not of CY.


CPU Design
RSR
RTR
IMM
OFF
SA
A_MUX
B_MUX
SB
A
B
CY
CY
ALU
OP
DIFF
RES
RSR
RTR
32 × 32 REG_FILE
RTA
WR
RSA
PC_IN
PC
PC_MUX
SPC
PC_OUT
PC_ADD
RSR
RSW
RSW_MUX
SSW
RES
S
L
ADDER_L
PC_L
PC_ADD_MUX
ADDER_1
PC_1
S_MUX
SS
DOUTB
PC_1
ADDRB
DOUTA
IR
DINB
WB
SAD
32 FAN-IN-OR
SPC_MUX
SC
ADDRA
DUAL PORT RAM
0
1
LZ
NZ
EZ
CY
CY
1
NOT
NOT
RSR[31]
DINA
WA
CONTROLLER
OP
DIFF
WA
WB
RTA
RSA
SA
SB
SSW
SS
WR
SC
SPC

Internal Structure of the ALU:

The ALU uses the following components:

- 32-bit Adder

  This takes in the values from the 32-bit lines A and B and produces the output A+B and updates the carry flag in register CY.

- 32-bit 2's complement

  This takes in a 32-bit value and produces the output as a 32-bit, two's complement of the input. It essentially inverts the individual bits as the numbers are stored in 2's complement notation.

- Logical And

  This takes in two 32-bit lines and computes the bit-wise logical 'and' of the bits and produces a 32-bit output.

- Logical Xor

  This takes in two 32-bit lines and computes the bit-wise logical 'xor' of the bits and produces a 32-bit output.

- 32-bit Decoder

  This component is used during the `diff` operation. It takes in a 32-bit input which has all but one bit zeroes. The output is a zero padded 32-bit number denoting the position (from 0 to 31) of the input which had the bit-value 1. In case all the bits of the input are 0 the output is the answer 32.

- Logical Shift Left

  This component takes in two 32-bit input lines and produces a 32-bit output obtained by shifting left the value of the first input by the value present in the second input.

- Logical Right Left

  This component takes in two 32-bit input lines and produces a 32-bit output obtained by shifting right (with zero-extension) the value of the first input by the value present in the second input.

- Logical Shift Left

  This component takes in two 32-bit input lines and produces a 32-bit output obtained by shifting right (with sign-extension) the value of the first input by the value present in the second input.

In order to increase efficiency the `diff` operation is implemented by reusing the components of the ALU by just adding multiplexers that select the necessary input depending upon the DIFF signal. The 32-bit final output, RES, is determined by the final 8×1 multiplexer depending upon the 3-bit control signal OP.

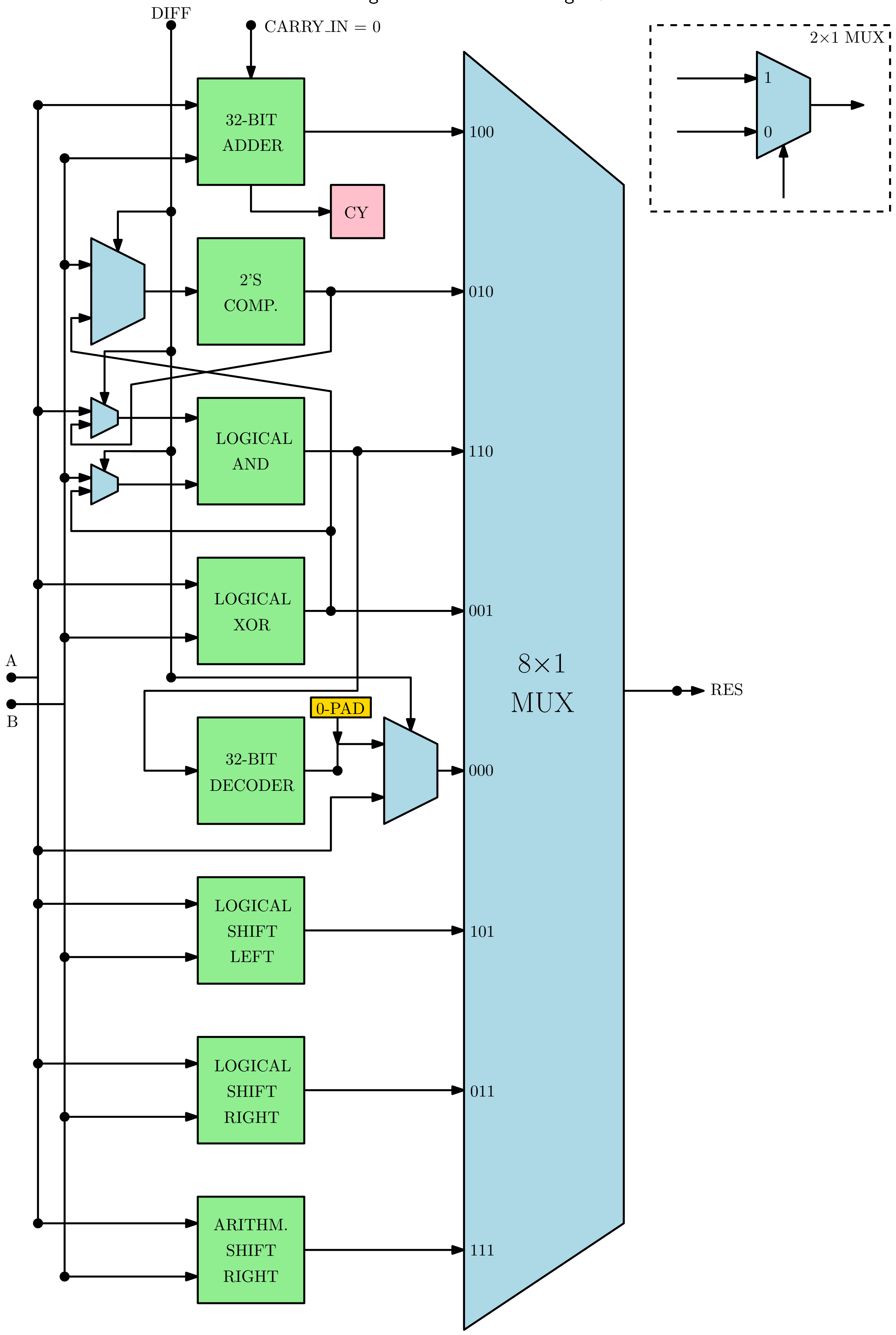
The `diff` operation is essentially used to compute the position of the first bit at which the two input operand registers differ. This is done by first taking a 'xor' of the operands. The binary value obtained from this 'xor' operation has 1 at the positions at which the two inputs differ. The bitwise 'and' between a number and its 2's complement is the largest power of 2 that divides the number itself. The largest power of 2 that divides a number is nothing but the number obtained by keeping the first bit (from the LSB) 1 and making the remaining bits zero. This suggests that we can take the output of the 'xor' we computed and take the bitwise 'and' between that value and its 2's complement. This would give the number which has 1 at the position where the original inputs differed. Passing this to the 32-bit decoder gives us the required answer which is the position at which the two inputs differed. The implementation is done by reusing the existing components and by adding multiplexers which suitably select the input lines depending upon the DIFF signal.

Specifications:

We have used a 4096×32-bit block RAM, initialised with a sorting program. The number of elements needed to be sorted is provided at location 31 and from 32^{nd} location, the numbers are present themselves.

We have used the inbuilt clock (of 4MHz) in Spartan3 FPGA board.

Internal Design of the Arithmetic Logic Unit



Control Signal Specifications:

The values for the control lines determined by the controller (control logic unit) for individual instructions are given in the below table.

Control Signals For Individual Instructions

INS	SAD	SS	SSW	WR	OP[0]	OP[1]	OP[2]	SPC	SC[2]	SC[1]	SC[0]	WB	SB[1]	SB[0]	SA
add	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
comp	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
addi	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0
compi	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
and	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0
xor	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
shll	0	0	0	1	1	0	1	0	0	0	0	0	1	0	0
shra	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0
shrl	0	0	0	1	1	1	1	0	0	0	0	0	1	0	0
shllv	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0
shrav	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
shllv	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
diff	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
lw	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1
sw	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1
b	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
br	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
bl	1	1	1	1	0	1	0	0	0	0	1	0	0	0	0
bltz	LZ	0	0	0	1	0	0	0	0	1	0	0	0	0	0
bz	~NZ	0	0	0	0	1	1	0	1	0	0	0	0	0	0
bnz	NZ	0	0	0	1	1	1	0	0	1	1	0	0	0	0
bcy	CY	0	0	0	1	0	1	0	1	0	1	0	0	0	0
bncy	~CY	0	0	0	1	1	0	0	1	1	0	0	0	0	0
nop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Important Features of the design:

- The encoding is in *little-endian* notation. Therefore the following shows how some instructions would be stored in the `coe` file.

  add (rs = 3) (rt = 6) $\longrightarrow$ 00000000000000000011000011000100

  addi (rs = 13) (imm = -15) $\longrightarrow$ 11111111111111111000101101100100

- The branch instructions are all *PC-relative*, except the `br` instruction, which transfers control to the absolute location stored in the register provided.
- For the `diff` instruction, we used the property that for any number c, $c\&(-c)$ gives us the highest power of 2 dividing c. Therefore to find the position of the least significant bit where a and b differ, we first find the highest power of 2 dividing $(a \oplus b)$ (which is $((a \oplus b)\&(-(a \oplus b)))$ and pass it through a *32-bit decoder* to get our final answer.
- There is provision to include more instructions (if necessary) by realising other combinations of `IR[5:3]`.