6037 CEM

Hardware and realisation of a computer system

Abdus Samad Saleem 11778028
6037 Hardware and realisation of a computer system

Contents

Introduction

This coursework is based on applying a variety of I type instructions and R type instructions on a single cycle processor, the I type instructions that have been implemented in this coursework are, xori, andi, ori and store byte. Each of these I type instructions are important for a processor. The purpose of Xori is used for bitwise operations with an immediate value, this instruction performs a XOR operation between a register value and the immediate value, the result of this operation is stored in the destination register. The importance of a xori is this allows XOR operations with immediate values, this is useful for data manipulations. The other I type instruction is ORI, this performs bitwise OR operation with an immediate value, the OR operation of a register value and a constant, the result of this is stored in the destination register. ORI are important as this is used for various data processing. Furthermore, the other I type instruction is andi, this instruction performs bitwise AND operation on each bit of a register with the corresponding bit of a constant, the result will be stored in the register destination. The purpose of andi is isolating certain fields of data. The final I type instruction that is used is sb (store byte), this instruction stores the least significant byte of a register into memory. This uses a base register, an offset, and the value in the source register which determines where the data is stored, the purpose of this is to store small data like character or bytes into memory. This is often used in I/O operations. In addition to this, the r type instructions that are used are also crucial, XOR performs the bitwise XOR operation in between the values in two registers, the result of this is stored in a destination register, this is used in a variety of applications one of them being data manipulation. The other R type instruction is SRL (shift right logical), this instruction shifts the bits of a register to the right by a specified number of positions, this instruction is useful in data scaling. The other r type instruction used is srlv (shift right logical variable) this instruction shifts the bits in a register to the right by a variable number of positions. This is useful when the number of positions shift depends on certain conditions. The last R type instruction used is JR (jump register) this is used for unconditional jumps, this transfer controls to the address contained in the register, the purpose of Jr is to implements functions returns by jumping to the address stored in the return address register.

Design

Instr	Opcode	Rs	Rt	Rd	Shamt	Funct	Hex
	31:26	25:21	20:16	15:11	10:6	5:0	
SRL	000000	00000	01111	11010	00011	000010	000FD0C2
			15	26	3		
Rd = rt >> shamt	000000	00000	00001	11010	00011	000010	0001D0C2
XOR: $d = $s ^ $t	000000	01011	01100	00111	00000	100110	016C3826
$d = $s ^ $t		11	12	7			
SRLV	000000	00100	11000	101011	00000	000110	01315806
		4	24	23			
$d = $t >> $s	000000	00100	00010	101011	00000	000110	01055806
Jr PC =RS	000000	01100	00000	000000	00000	001000	03000008

This is the R-type instructions and the values of rs and rt, opcode for R type instructions is 0. Each function code for the r type is unique to each instruction. The shamt field is 0 for xor, srlv and jr. the rs in srl must be 0, this is because the operation of srl doesn't require the first sourse register. Furthermore, for Jr the only register required is rs.

R-type instructions

ALUOP 2:0	Funct	Alucontrol 4:0
110	100000	00010 (ADD)
110	100010	00110 (SUB)
110	100100	00000 (AND)
110	100101	00001 (OR)
110	101010	00111(SLT)
110	000001	00100(SRL)
110	100110	00101(XOR)
110	000110	01100(SRLV)
110	001000	01000 (JR)

The aluop is the same for r type as the control bit for r type instructions are the same as the control bit is the same for r type. The alu control is increased from 4 bits to 5 bits to allow more instructions to have an alu control bit. This will enable the instruction to operate correctly, due to the fact that not more than one instruction can have the same alu control value.

xor

	Opcode	Rs	Rt	RD	shamt	funct		
xor $7, $11 $12	000000	01011	01100	00111	00000	001101		016C380D
Addi $7, $11 10		Rs		Rd		Imm 10		Hex
001000		01011		00111		0000000000001001		21670009

This is the encoding for xor, here the addi adds a constant to the value and stores the value in the destination register. The constant here that is being added is 10, the destination register is 7, the first source register is 12 and register target is 11, so here the bitwise xor operation will be rs and rt and the value of that is stored in rd.

Instr	Opcode	Rs	Rt	Rd	Shamt	Funct	Hex
	31:26	25:21	20:16	15:11	10:6	5:0	
SRL	000000	00000	01111	11010	00011	000010	000FD0C2

From the table above the rt value is 15 and the value of shamt is 3, this operation will shift the value in rt 3 times to the right replacing the values that are shifted with 0, as shown on the encoding below. The addi instruction will add the immediate value 19 with the contents in the register rs which is 2, the value of this will be stored in rd.

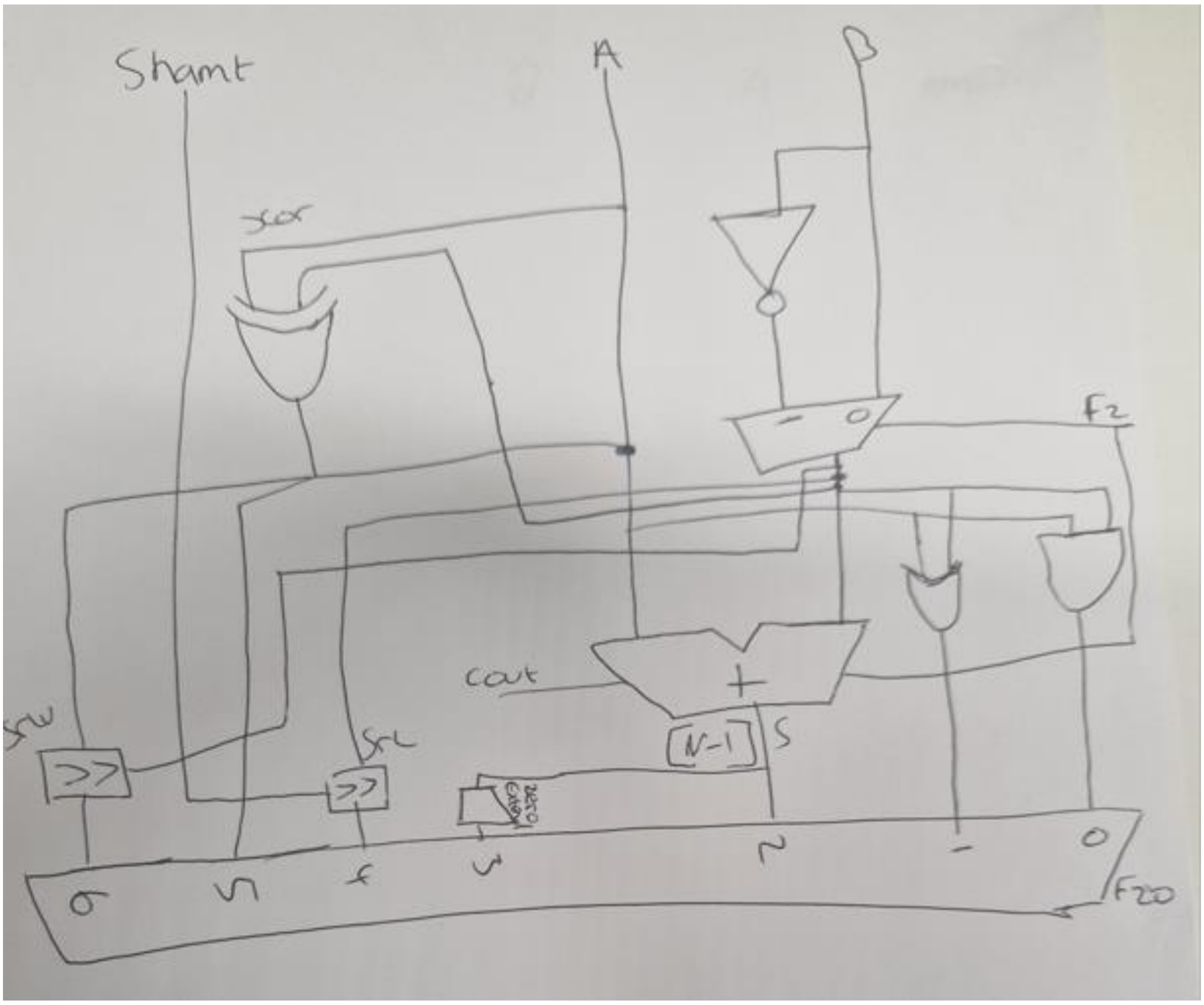
	Opcode	Rs	Rt	Rd	Shamt	funct
srl $26= $2 << $3	000000	00010	00000	11010	00011	000010
Addi $26, $2 19		Rs	Rd	Imm 19	Hex	
001000		00010	11010	0000000000010011	205A0013	

SRLV	000000	00100	11000	101011	00000	000110	01315806
		4	24	23			
$d = $t >> $s	000000	00100	00001	101011	00000	000110	01035806

So, the table above shows the operation of srlv, the value is rs determines how many times the value in rt must be shifted to the right, so here the value in rs is 4, and the value of rt is 24, after the operation the value of rt is 1, the table below shows the result after this operation. The addi will add the immediate value which is 25, then will be added to the contents in rs 4, then the result of this will be stored in rd.

	Opcode	Rs	Rt	Rd	Shamt	funct
srlv $23 = $1 >> $4	000000	00100	00110	101011	00000	000110

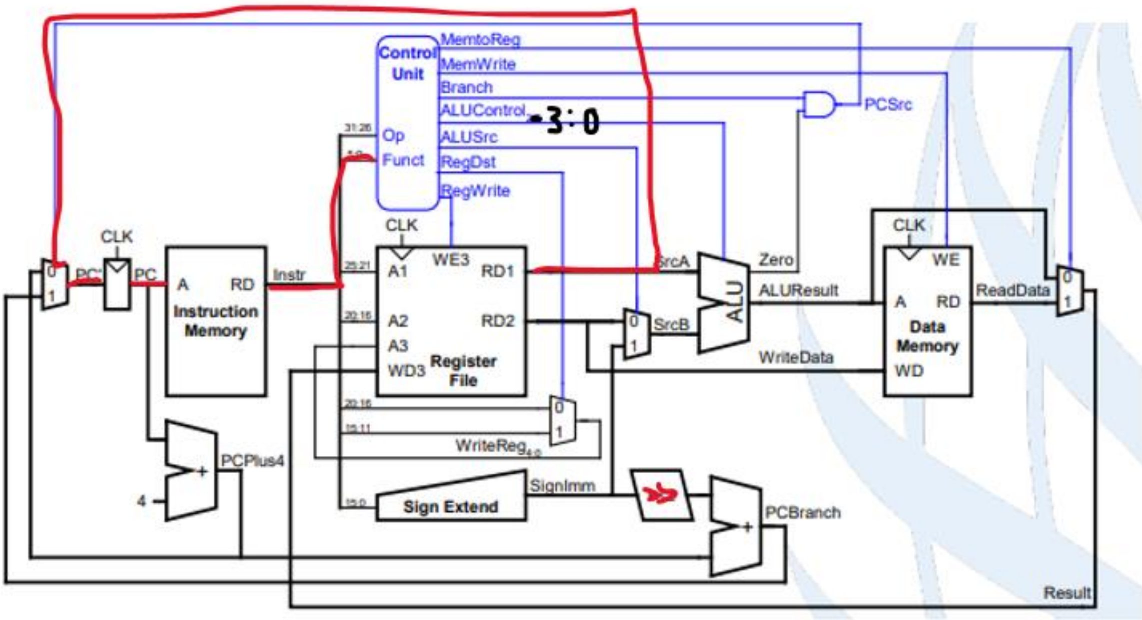
Addi $23, $1 25	Rs	Rd	Imm 25	Hex
001000	00100	101011	0000000000011001	412B0019



JR

	Opcode	Rs	Rt	Rd	Shamt	funct	Hex
PC = R[rs] $9	000000	01001	00000	00000	00000	001000	01200008

Here the jump register will set the program counter to the value in register 9. This instruction will cause a jump an unconditional jump to the address stored in register 9.



I TYPE FUNCTION

Instr	Opcode	Rs	Rt	Immediate	hex
	31:26	25:21	20:16	15:0	
XORI	001110	01011	01110	0000000000001111	396E000F
Rs,rt,imm		11	14	15	
Ori	001101	10011	10111	0000000000011111	3677001F
$t =$s|imm		19	23	31	
ANDI	001100	10110	11110	0000000000101010	32DE002A
[rt] = [rs] & ZeroImm		22	30	42	
SB rt, rs imm	101000	01101	10101	0000000000010110	A1B50016
		13	21		

Here is the list of I type instructions that have been implemented in this coursework, the above table show the operations of each instruction.

I-type instructions

ALUOP 2:0	Funct	Alucontrol 3:0
000	X	0010 (ADD)
010	X	0110 (subtract)
001	X	0111(ORI)
011	X	1000(XORI)
100	X	1001(SB)
101	X	1111(ANDI)

The table above shows the ALUop for each instruction, the purpose of this is that each I type instruction the ALUop is unique, the bit width has increased from 2:0 to 3:0 allowing all the values in the instructions to have their own unique alucontrol value. I type instructions they don't have functions code.

Instruction	Regwrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	jump	AluOp
I-Type								
Xori	1	0	1	0	0	0	0	011
Ori	1	0	1	0	0	0	0	001
Andi	1	0	1	0	0	0	0	101
SB	0	0	1	0	1	0	0	111

For I type instructions as they have their own unique alucontrol number, it is essential to have their own control bit, the purpose of each instruction to have their own control bit shows how the instructions operate on a single cycle mips processor data path.

Ori

	Opcode	Rs	Rt	Imm	
Ori $23, $19 31	001101	10011	10111	0000000000011111	
Addi $23, $19 31	001000	10011	10111	0000000000011111	2277001F

Here is the encoding for ori, the $23 is rt where the results of the bitwise OR operation will be stored, $19, is the source register this contains one of the operands for the OR operation. The immediate value 31 and the value in $19 will be Ored.

xori

	Opcode	Rs	Rt	Imm	
xori $14, $11 15	001110	01011	01110	0000000000001111	
Addi $14, $11 15	001000	01011	01110	0000000000001111	216E000F

The table above shows the encoding for xori, the $14 is rt, in $14 the value of the result is stored here. The register $11 is the source register, here the immediate value is 15. This means that 15 will be xored with value in register $11.

andi

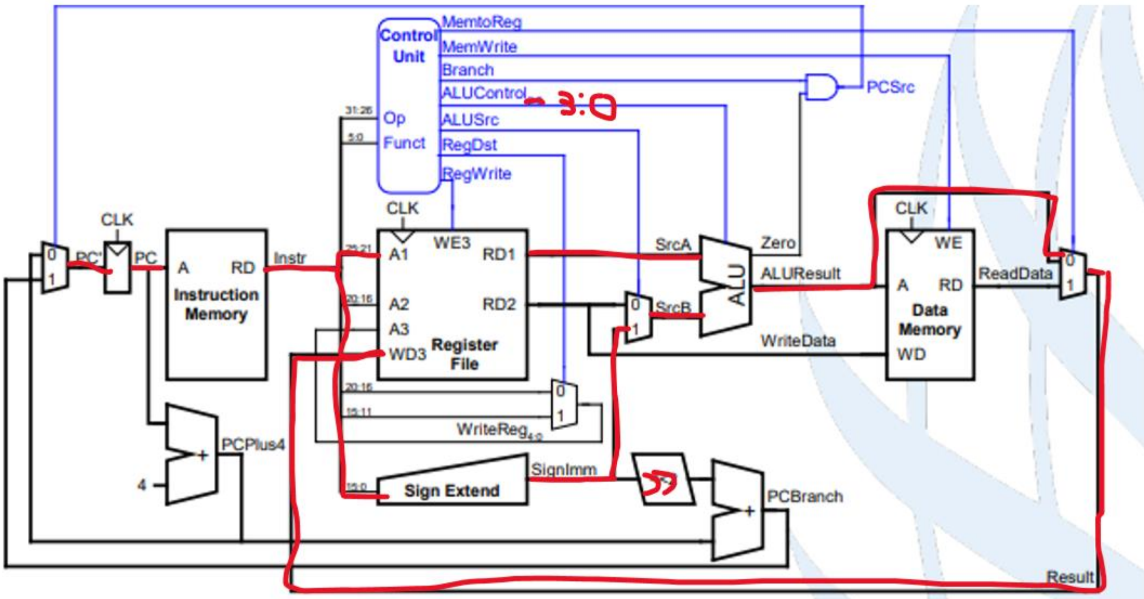
	Opcode	Rs	Rt	Imm	
andi $30, $22 42	001100	10110	11110	0000000000101010	
Addi $30, $22 42	001000	10110	11110	0000000000101010	22DE002A

In register $30, this is the destination register where the result of the AND operation will be stored. In register $22 is the source register, and here the immediate value is 42, so the AND operation will be 42 with the contents in $22. The purpose of the addi instruction is to add values to rt.

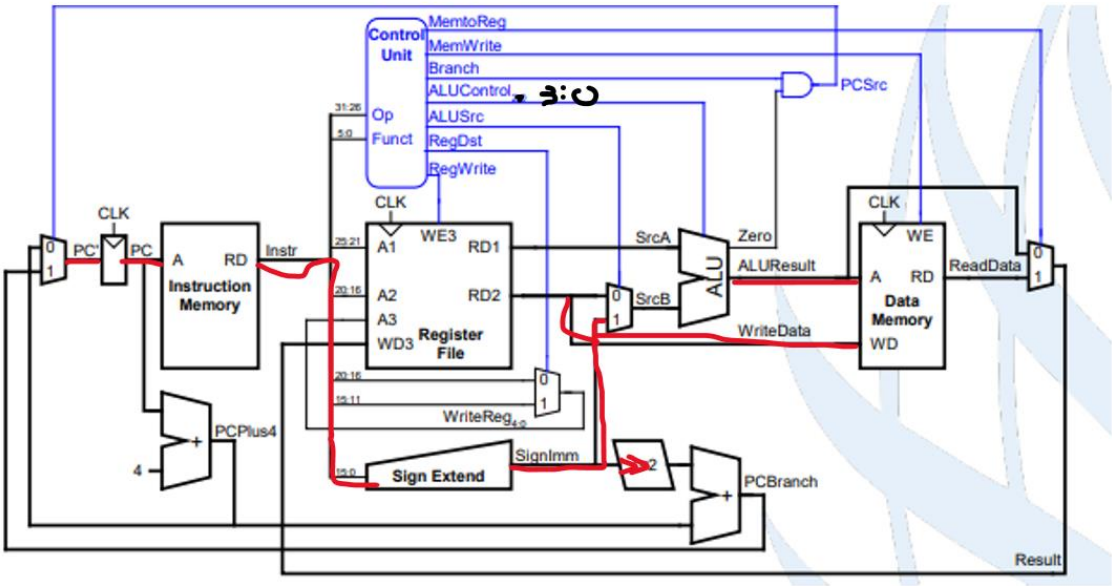
SB

	Opcode	Rs	Rt	Imm	
Sb $21, 17($13)	101000	01101	10101	0000000000010001	
Addi $21, $13 17	001000	10110	11110	0000000000010001	22DE0011

The source register $21, the least significant bit will be stored in memory, $13 is the base register, this specifies the base address in memory. The immediate offset 17, this determines from the base address where the byte will be stored.



Here is the datapath for andi, xori and ori, as shown on the data path xori, ori and andi have the same data path, however the control signals are different, this is shown on the control bit table, how each I instruction have their own unique ALU control value.



This is the data path for SB, as shown this is different to the data path above as not only the ALU control value is different the control unit signals are different as well as SB is different to xori, ori and andi.

Design code

```
always_comb
  case(op)
    6'b000000: controls <= 10'b110000010; // RTYPE
    6'b100011: controls <= 10'b101001000; // LW
    6'b101011: controls <= 10'b001010000; // SW
    6'b000100: controls <= 10'b000100001; // BEQ
    6'b001000: controls <= 10'b101000000; // ADDI
    6'b000010: controls <= 10'b000000100; // J
    6'b001110: controls <= 10'b1010000011; // xori
    6'b001100: controls <= 10'b1010000101; // andi
    6'b001101: controls <= 10'b1100100001; // ori
    6'b101000: controls <= 10'b0010100111; // sb
```

As shown above the design code shows that r-type instructions don’t have an opcode, so all of their op code is the same and assigned to 6’b000000, and they control unit is assigned to 10’b110000010.

For the instructions that have been implemented which are xori, andi, ori, sb. These instructions have they own unique aluop code and control unit code. As shown on the table in chapter 2, the control bit is determined from the data path which shows what each instruction is executing on the single cycle processor. In addition to this the instruction sb has a different function from xori, andi, and ori. As the purpose of this instruction is different so the control bit will be different.

```
always_comb
  case(aluop)
  2'b00: alucontrol <= 4'b0010;// add (for lw/sw/addi)
  2'b01: alucontrol <= 4'b1010; //sub for beq
fault: case(funct)          // R-type instructions

      6'b100000: alucontrol <= 4'b0010; // add
      6'b100010: alucontrol <= 4'b1010; // sub
      6'b100100: alucontrol <= 4'b0000; // and
      6'b100101: alucontrol <= 4'b0001; // or
      6'b101010: alucontrol <= 4'b1011; // slt
      6'b100110: alucontrol <= 4'b0101; // XOR
      6'b000000: alucontrol <= 4'b0100; // srl
      6'b000110: alucontrol <= 4'b0110; // srlv
      6'b001000: alucontrol <= 4'b1111; // jr
      default:   alucontrol <= 4'bxxxx; // ???
    endcase
  endcase
```

The code above shows the 6-bit value is the function code for each instruction, as mentioned before each r-type instruction have they own unique function code. In addition to this the 4-bit binary value is the alu control value for each instruction. The reason as to why the value is 4 bits, this is because the number of R type instructions that are implemented are required to have their own unique alu controls values, so increasing the bit size will help with this operation.

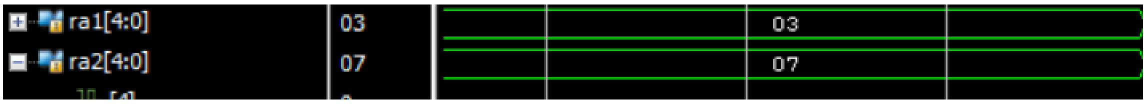
```
logic [31:0] rf[0:31];
assign rf[11] = 32'b01011; // REGISTER L
assign rf[12] = 32'b01100; // xor
assign rf[2]  = 32'b00010; // srl rt
assign rf[14] = 32'b00100; // srlv rs
assign rf[15] = 32'b00110; // srlv rt
assign rf[17] = 32'b01001; // jr
```

The code above shows the binary value that is assigned to the first register source and the register target. These values are assigned to the register location in rf, the purpose of this is to allow the operation of each instruction to operate and store that value in the destination register.

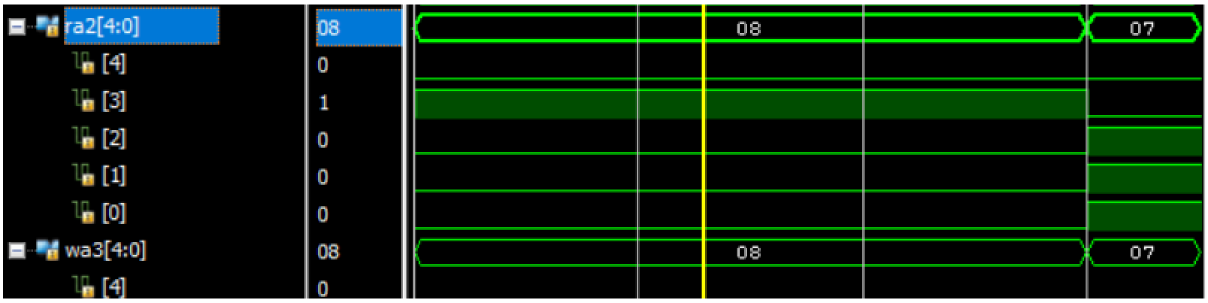
```
  3'b000: result = a & condinvb; //AND
  3'b001: result = a | condinvb;//OR
  3'b010: result = sum;
  3'b011: result = sum[31];
  3'b100: result = condinvb >> shamt; //SRL
  3'b101: result = condinvb >> a; //SRLV
  3'b111: result = a ^ condinvb; //XOR
endcase
```

The code above shows the bitwise operation for xor, srl, and srlv. The use of “>>” ensures that the shift right logical is assigned to perform this action, furthermore for srl and srlv the number of times to shift to the right is instructed by the value of shamt for srl, and the value of rs for srlv. . The xor operator is “^”, this performs the xor of rt and rs.

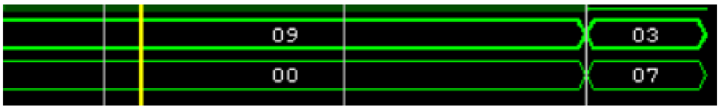
Verification



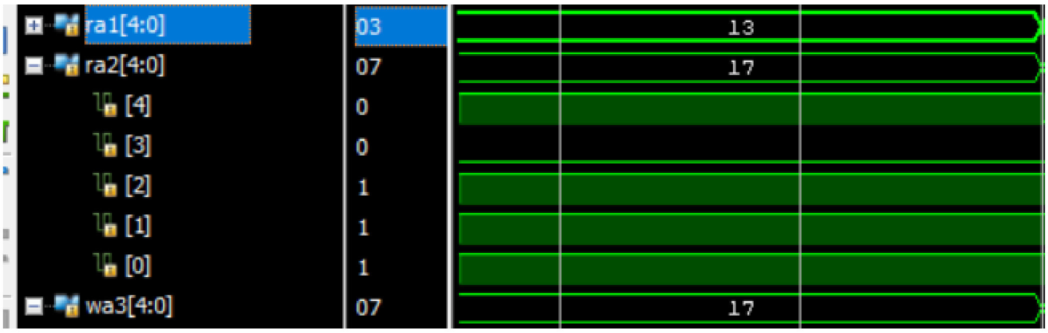
The value 7 here on ra2 shows the bitwise operation for xor, and the value 03 shows the srl operation. According to the xor truth table the result is 7, and for srl the value should be 3 as stated above the shift amount is 3.



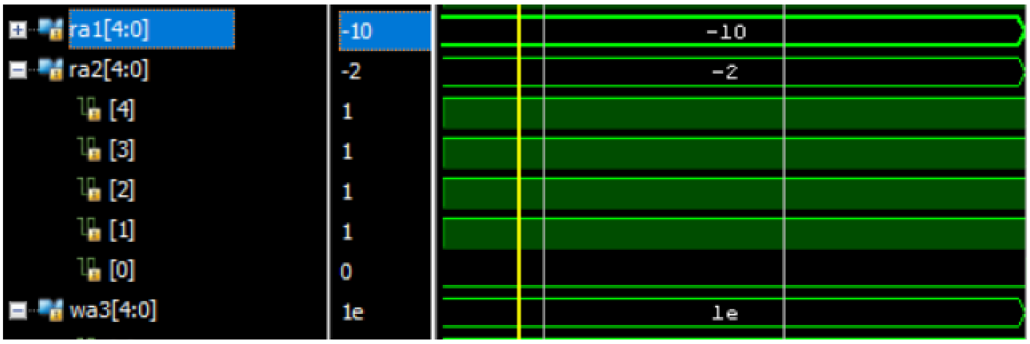
The value of the shift right logical variable is stored in register 8, as shown above the value of srlv is shown in register 8. This shows that the memory file is accurately reading the values to the correct destination register.

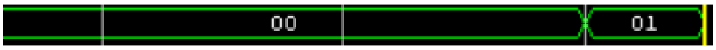


Here for jump register the value of rs is 9, so this shows that the value of the new program counter is stored in register 9. The above shows that register 9 is signed and the value in register 9 is the value of the new program counter.



For ori, andi and xori the value of the new result is stored in the destination register that is stated on the encoding, the encoding is crucial as this provides a hexadecimal number to place on the memfile. The purpose of this is to allow the values that are required in andi, ori and xori to be presented on the wavelength, and stored in the correct register that is specified in the encoding.





The numbers above show the least significant byte that is stored in the memory location specified in the encoding for sb. This shows that the hexadecimal value that has been calculated for sb is stored in the source register where the least significant byte is stored in the memory.

Conclusion

To conclude I type instructions and R type Instructions are crucial instructions as each have their own characteristics. The key point of I type instructions are designed to for operations that have immediate values. Their control signals are unique for memory access, ALU operations and data movement. The purpose of R type instruction is used for operations involving data in registers. Overall, the combination of I-type and R-type instructions provides adaptability, this means a board variety of operations. As the R type instructions was used to implement a single cycle mips processor, as each instructions takes one clock cycle to complete their task. Furthermore, for I type instruction the control bit has to be accurate for the instruction to perform their operations. In addition to this R type instruction do not involve memory access, they only operate with data that is stored in registers.