King Fahd University of Petroleum and Minerals
College of Computer Science and Engineering
Computer Engineering Department

COE 301 COMPUTER ORGANIZATION
ICS 233: COMPUTER ARCHITECTURE & ASSEMBLY LANGUAGE

Term 171 (Fall 2017-2018)
Major Exam 2
Saturday Dec. 2, 2017

Time: 150 minutes, Total Pages: 14

Name:_____ ID:_____ Section: _____

Notes:

- Do not open the exam book until instructed
- Answer all questions
- All steps must be shown
- Any assumptions made must be clearly stated

Question	Max Points	Score
Q1	18	
Q2	18	
Q3	10	
Q4	24	
Total	70	

Dr. Aiman El-Maleh
Dr. Marwan Abu Amara

[18 Points]

(Q1)

(i) (10 points) A **palindrome** text is a text that reads the same backward or forward. For example, both **abcdcba** and **abcddcba** are palindrome texts. Write a **recursive** MIPS procedure that implements the following high-level **palindrome** procedure code using minimal instructions. The **palindrome** procedure should test if a text is palindrome or not. Assume the text to be tested is currently in the memory. Assume further that before calling the **palindrome** procedure, the **$a0** and **$a1** registers already contain the memory addresses of the first and the last characters of the text held in the memory, respectively. The **palindrome** procedure returns **1** in register **$v0** if the text is palindrome, and returns **0** in register **$v0** if the text is not palindrome. Use MIPS programming convention in saving and restoring only the necessary register(s) in the procedure. Note that the variables **first** and **last** in the high-level procedure code refer to the memory locations of the first and the last characters of the text, respectively. On the other hand, ***first** and ***last** refer to the contents of the memory locations held in the variables **first** and **last**, respectively.

```
int palindrome(int first, int last) {
      if ((last – first) < 1) return 1;
      else if (*last != *first) return 0;
      else return (palindrome(++first, --last));
}
```

```
palin:
      subu  $t1,$a1,$a0  # $t1 = (end - begin)
      slti  $t0,$t1,1    # $t1 < 1 ? If true, set $t0 to 1, else 0
      beqz  $t0,elseif   # (end - begin) < 1 ?
      li    $v0,1        # return 1 (true)
      jr    $ra
elseif:
      lbu   $t0,0($a0)   # get char at the beginning
      lbu   $t1,0($a1)   # get char at the end
      beq   $t0,$t1,else # *(begin) != *(end) ?
      move  $v0,$zero    # return 0 (false)
      jr    $ra
else:
      addiu $sp,$sp,-4   # save $ra
      sw    $ra,0($sp)
      addiu $a0,$a0,1    # ++begin
      addiu $a1,$a1,-1   # --end
      jal   palin        # palin(++begin, --end)
      lw    $ra,0($sp)
      addiu $sp,$sp,4    # save $ra
      jr    $ra
```

(ii) (8 points) Assume that a procedure **f** calls another procedure **g** twice as shown in the given high-level code. Procedure **g** expects two **signed** integers to be passed to it as parameters in registers **$a0** and **$a1**, and returns a **signed** integer as a result in **$v0**. It is not known what **g** does, or which registers are modified by **g**. Assume that the values of **a**, **b**, and **c** are not needed by the procedure calling the procedure **f**. Assume further that before calling procedure **f**, registers **$a0** = **a**, **$a1** = **b**, and **$a2** = **c**, where **a**, **b**, and **c** are **signed** integers. Procedure **f** returns the **signed** integer result in **$v0**. Write a MIPS procedure that implements **f**. Use MIPS programming convention in saving and restoring only the necessary register(s) in the procedure.

```
int f(int a, int b, int c) {
      if (a > c) {
            int d = g(a, g(a, c));
      }
      else {
            int d = g(c, g(a, c));
      }
      return (b + d);
}
```

```
f:
      addiu $sp,$sp,-12  # frame = 12 bytes
      sw    $ra,0($sp)   # save $ra
      sw    $a1,4($sp)   # save argument b
      move  $a1,$a2      # set 2nd arg. for 1st call to g to be c

      slt   $t0,$a2,$a0  # (a > c) ? If true, set $t0 to 1, else to 0
      beqz  $t0,else     # If $t0 = 0, then (a ≤ c) and branch to else
      sw    $a0,8($sp)   # save argument a
      jal   g            # call g(a,c) -- 1st call to g
      lw    $a0,8($sp)   # set 1st arg. for 2nd call to g to be a
      move  $a1,$v0      # set 2nd arg. for 2nd call to g to be g(a,c)
      jal   g            # call g(a, g(a,c)) -- 2nd call to g
      j     return       # go to return code to compute (b + d)
else:
      sw    $a2,8($sp)   # save argument c
      jal   g            # call g(a,c) -- 1st call to g
      lw    $a0,8($sp)   # set 1st arg. for 2nd call to g to be c
      move  $a1,$v0      # set 2nd arg. for 2nd call to g to be g(a,c)
      jal   g            # call g(c, g(a,c)) -- 2nd call to g
return:
      lw    $t0,4($sp)   # restore b
      addu  $v0,$t0,$v0  # return $v0 = (b + d)
      lw    $ra,0($sp)   # restore $ra
      addiu $sp,$sp,12   # free stack frame
      jr    $ra          # return to caller
```

[18 points]

(Q2)

(i) (3 points) Find the **decimal value** of the following single-precision float:

S	Exponent	Fraction
1	1000 1011	000 0100 1100 1100 0000 0000

Sign bit = 1 (negative)
Biased Exponent = 1000 1011 = 139
Exponent Value = 139 – 127 = +12
Value = $-(1.000\ 0100\ 1100\ 1100\ 0000\ 0000)_2 \times 2^{+12}$
$= -(1000010011001.100\ 0000\ 0000)_2$
Decimal Value = -4249.5

(ii) (3 points) Find the **normalized** IEEE 754 single-precision representation of **-21.40625**.

-21.40625 = -10101.01101

Normalize: $-21.40625 = -1.010\ 1011\ 0100\ 0000\ 0000\ 0000 \times 2^{+4}$

S	Exponent	Fraction
1	1000 0011	010 1011 0100 0000 0000 0000

(iii)(5 points) Normalize and Round the given single-precision number with given GRS (Guard, Round, and Sticky) bits using the following four rounding modes. Show the **final normalized** number and its exponent:

GRS
+0.111 1111 1111 1111 1111 1111 110 × 2^{+10}

Normalize: +1.111 1111 1111 1111 1111 1111 100 × 2^{+9}

Round towards Zero: +1.111 1111 1111 1111 1111 1111 × 2^{+9}

Round towards +Infinity: +1.000 0000 0000 0000 0000 0000 × 2^{+10}

Round towards -Infinity: +1.111 1111 1111 1111 1111 1111 × 2^{+9}

Round towards Nearest Even: +1.000 0000 0000 0000 0000 0000 × 2^{+10}

(iv)(7 points) Given that **A** and **B** are single-precision floats, compute the difference **A−B**. Use rounding to **nearest even**. Perform the operation using **guard**, **round** and **sticky** bits.

A = +1.011 1001 0101 0000 0011 0000 × 2^{-3}
B = +1.111 1010 0011 0101 0111 1111 × 2^{+2}

	1.011 1001 0101 0000 0011 0000 000 x 2^{-3}	
−	1.111 1010 0011 0101 0111 1111 000 x 2^{+2}	
	00.000 0101 1100 1010 1000 0001 100 x 2^{+2}	**(align)**
−	01.111 1010 0011 0101 0111 1111 000 x 2^{+2}	
	00.000 0101 1100 1010 1000 0001 100 x 2^{+2}	
+	10.000 0101 1100 1010 1000 0001 000 x 2^{+2}	**(2's complement)**
	10.000 1011 1001 0101 0000 0010 100 x 2^{+2}	
= −	1.111 0100 0110 1010 1111 1101 100 x 2^{+2}	
= −	1.111 0100 0110 1010 1111 1110 x 2^{+2}	**(round)**

[10 Points]

(Q3)

(i) **(4 points)** Given that **Multiplicand=0111** and **Multiplier=1011,** using the **signed multiplication** hardware, show the **signed** multiplication of **Multiplicand** by **Multiplier**. The result of the multiplication should be an 8-bit **signed** number in HI and LO registers. Show the steps of your work.

Iteration		**Multiplicand**	**Sign**	**Product = HI,LO**
0	Initialize	0111		0000 1011
1	LO[0] = 1 => ADD		0	0111 1011
	Shift Product = (HI, LO) right 1 bit	0111		0011 1101
2	LO[0] = 1 => ADD		0	1010 1101
	Shift Product = (HI, LO) right 1 bit	0111		0101 0110
3	LO[0] = 0 => Do nothing		0	0101 0110
	Shift Product = (HI, LO) right 1 bit	0111		0010 1011
4	LO[0] = 1 => SUB (ADD 2's compl)	1001	1	1011 1011
	Shift Product = (HI, LO) right 1 bit			1101 1101

(ii) **(6 points)** Given that **Dividend=1001** and **Divisor=0010** are signed 2's complement numbers, show the **signed** division of **Dividend** by **Divisor**. The result of division should be stored in the Remainder and Quotient registers. Show the steps of your work, and show the final result.

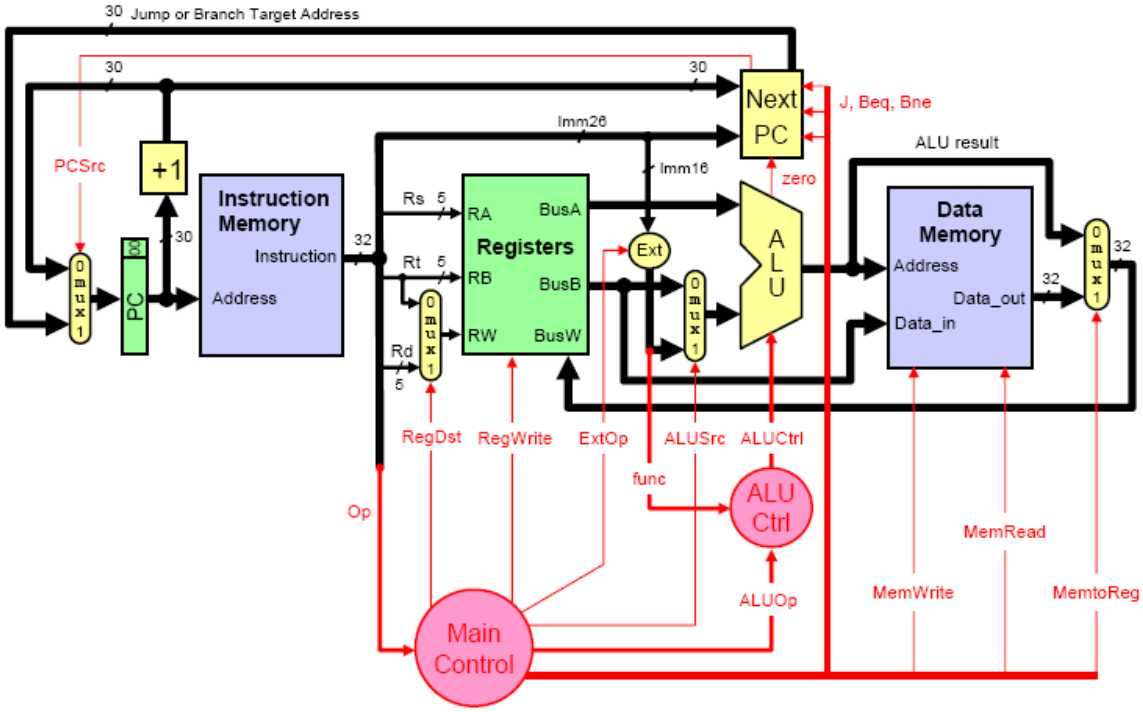
Since the Dividend is negative, we take its 2's complement ⇒ Dividend = 0111
Sign of Quotient = negative, Sign of Remainder = negative

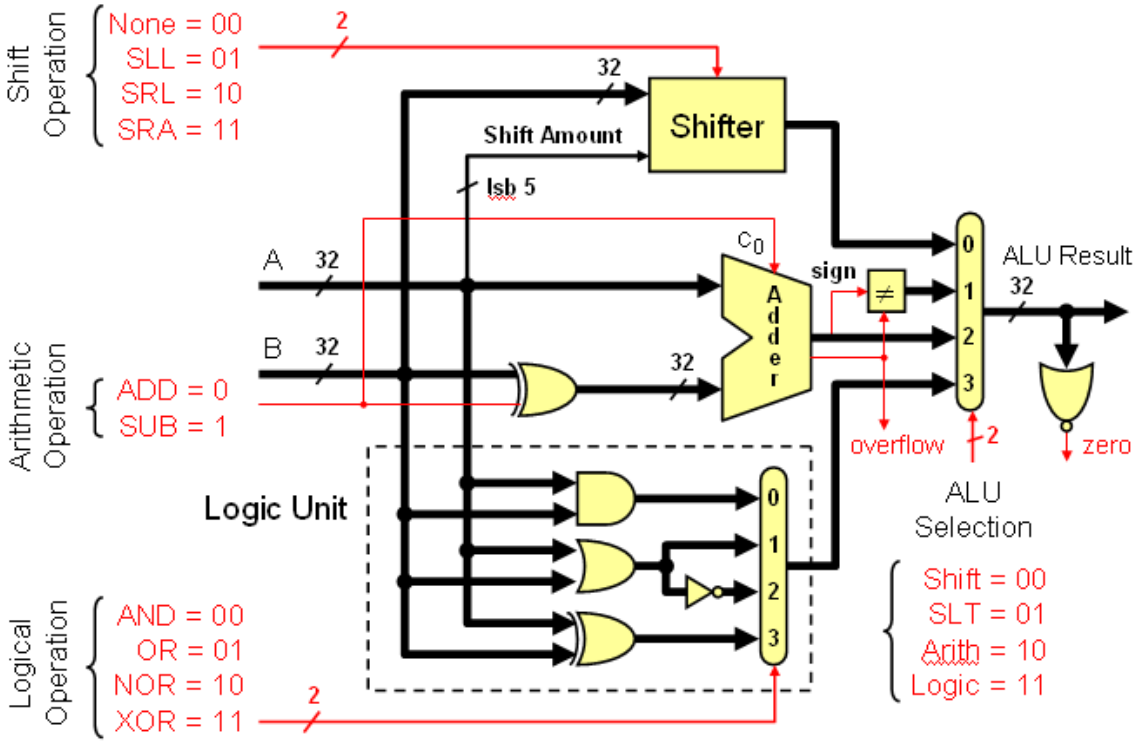
Iteration		**Remainder (HI)**	**Quotient (LO)**	**Divisor**	**Difference**
0	Initialize	0000	0111	0010	
1	1: SLL, Difference	0000	1110	0010	1110
	2: Diff < 0 => Do Nothing	0000	1110	0010	
2	1: SLL, Difference	0001	1100	0010	1111
	2: Diff < 0 => Do Nothing	0001	1100	0010	
3	1: SLL, Difference	0011	1000	0010	0001
	2: Rem = Diff, set lsb Quotient	0001	1001	0010	
4	1: SLL, Difference	0011	0010	0010	0001
	2: Rem = Diff, set lsb Quotient	0001	0011	0010	

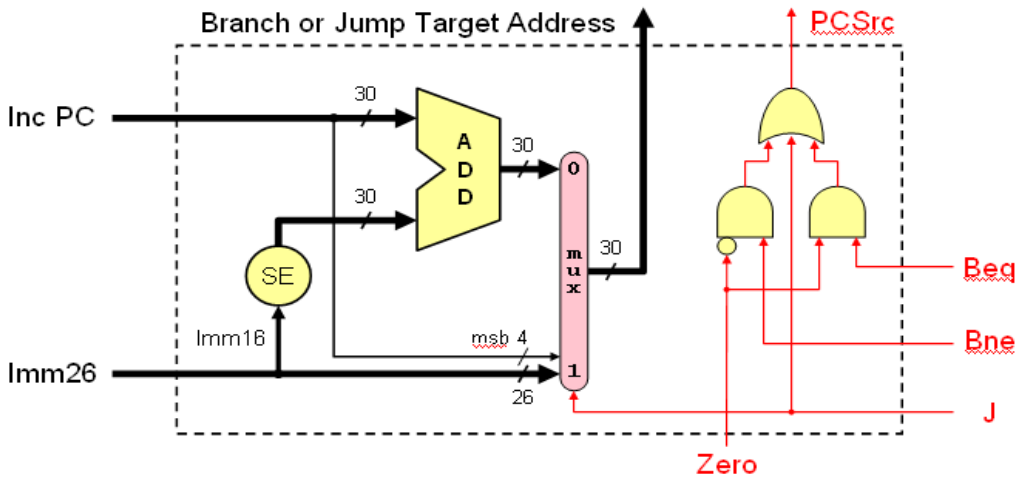
Quotient = 1101 **Remainder** = 1111

[24 Points]

(Q4) Consider the single-cycle datapath and control given below along with ALU and Next PC blocks design for the MIPS processor implementing a subset of the instruction set:







Details of Next PC

(i) **(5 points)** Show the control signals generated for the execution of the following instructions by filling the table given below:

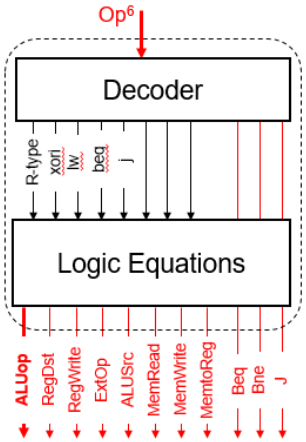
Op	RegDst	RegWrite	ExtOp	ALUSrc	ALUOp	Beq	Bne	J	MemRead	MemWrite	MemtoReg
R-type	1 = Rd	1	x	0=BusB	R-type	0	0	0	0	0	0
xori	0 = Rt	1	0=zero	1=Imm	XOR	0	0	0	0	0	0
lw	0 = Rt	1	1=sign	1=Imm	ADD	0	0	0	1	0	1
bne	x	0	x	0=BusB	SUB	0	1	0	0	0	x
j	x	0	x	x	x	0	0	1	0	0	x

(ii) **(4 points)** Show the block diagram for designing the control unit for this CPU and show the logic gates or equations for the control signals **RegDst**, **RegWrite** and **ExtOp** based on these instructions. Assume that the opcode of these instructions is a 6-bit opcode such that the opcode for R-type instructions is 0, the opcode for xori is 1, the opcode for lw is 2, and so on for the rest of the instructions.

RegDst <= R-type

RegWrite <= $\overline{(bne + j)}$

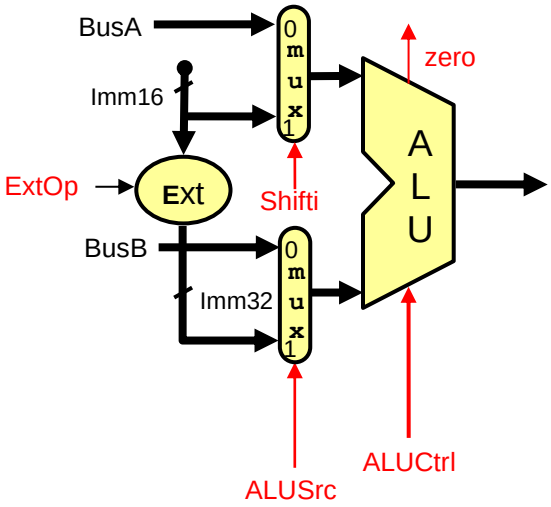
ExtOp <= lw



(iii) **(12 points)** We wish to add the following instructions to the MIPS single-cycle datapath. Add any necessary datapath modifications and control signals needed for the implementation of these instructions. Show only the **modified** and **added** components to the datapath.

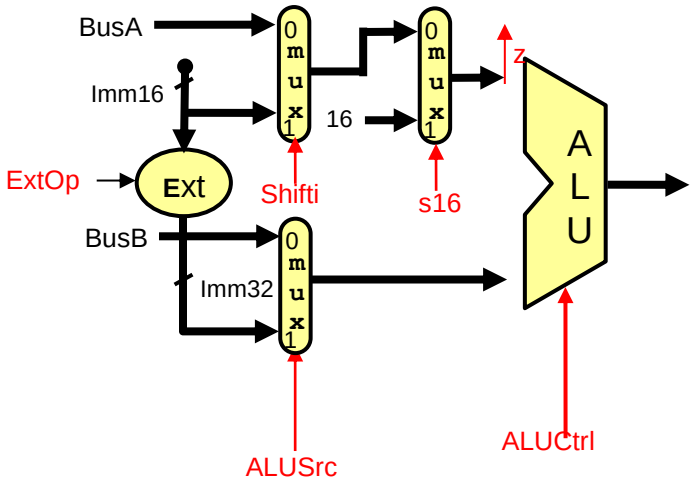
a. srl

For the srl instruction, examining the ALU one can see that the shift amont is coming through the A-input of the ALU and the operand to be shifted comes through the B input of the ALU. Thus, we need-to add a MUX on the A-input to select between the output of a register and the immediate values. This MUX needs to select only between the least significant 5 bits of BusA and bits 6 to 10 from Imm16. The modified part in the datapath is shown below:



b. lui

For this instruction, the shift amount is 16 and the operand to be shifted is the immediate value selected on the B-input of the ALU. Thus, we need to add another MUX to select the shift amount as 16 for this instruction. The modified parts of the datapath to support the execution of this instruction is given below:

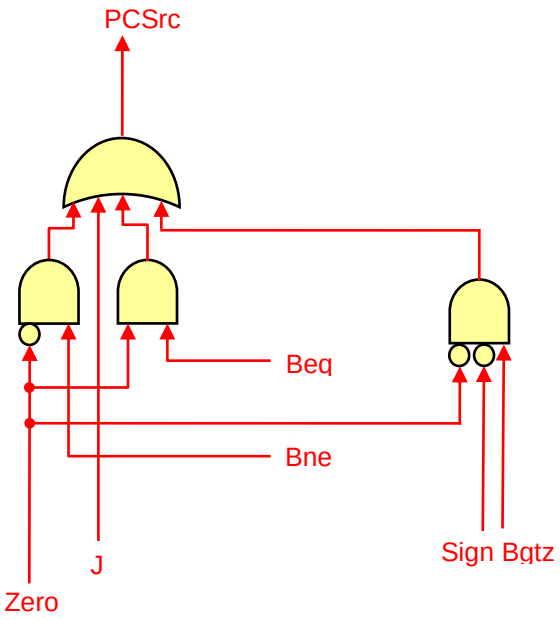


c. bgtz

Since the first source operand specified by RS comes on BusA and the second operand which is the Zero register specified by the RT filed comes on BusB, all we need is to get the operand on BusA to appear at the output of the ALU as we just need to check the sign bit (i.e. most significant bit of the result).

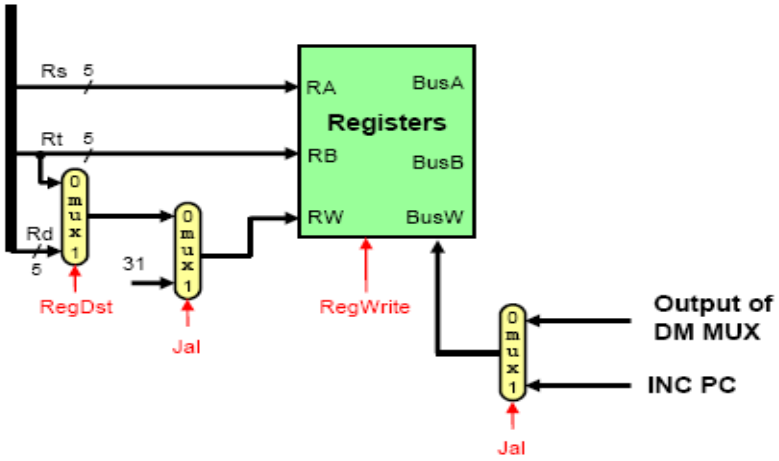
Performing an addition, subtraction, xoring, oring operations will work. Let us assume that we will do an ALU addition operation.

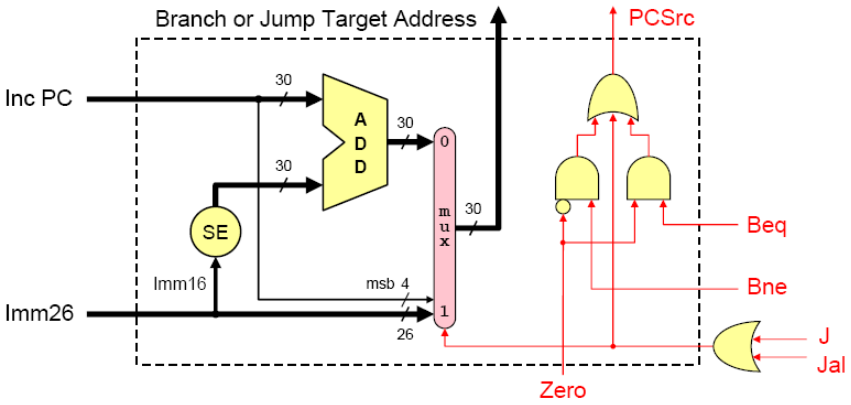
To check that the result is greater than 0, we need to check that the sign bit is 0 and that the result is not equal to zero. Thus, the changes needed to be done are in the NextPC block as shown below:



d. jal

This instruction is similar to the jump instruction (J) with the difference that register $31 should be loaded with the incremented PC value. Thus, we need to add a MUX at the input of RW input to the register file to select the value 31 when executing this instruction. We also need to add a MUX at the input of BusW in the register file to select the incremented PC value to be loaded instead of the value coming from the output of the data memory MUX. In addition, we need to make changes to the NextPC block to perform the same operation needed by the J instruction for Jal instruction. These changes are shown below:





e. lwi; This is a new instruction with the following format: lwi Rt, imm^{16}. Rt = MEMORY[imm^{16}]. Assume that imm^{16} will be zero extended.

For this instruction, the address for memory access should be loaded from the immediate value. Thus, we need to add a MUX at the input of the Data Memory unit to select between the ALU output and the output of the Extend unit.

(iv) **(3 points)** Suppose that you are asked to reduce the clock cycle of this CPU. What changes in the instruction set and the resulting CPU design you would do to achieve this goal? Clearly explain your solution and show the added changes in the CPU circuit design.

We will change the format of the load and store instructions so that the address is specified by a register only without a displacement. This will make the data memory unit to be accessed directly after the register file access without the need to wait for the ALU to finish. Thus, the changes in the design will be having the Data Memory with inputs for the address input connected to BusA instead of the ALU and the data input to remain connected to BusB.

MIPS Instructions:

Instruction	Meaning	R-Type Format					
add $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x20
addu $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x21
sub $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x22
subu $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x23

Instruction	Meaning	R-Type Format					
and $s1, $s2, $s3	$s1 = $s2 & $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x24
or $s1, $s2, $s3	$s1 = $s2 | $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x25
xor $s1, $s2, $s3	$s1 = $s2 ^ $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x26
nor $s1, $s2, $s3	$s1 = ~($s2|$s3)	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x27

Instruction	Meaning	R-Type Format					
sll $s1,$s2,10	$s1 = $s2 << 10	op = 0	rs = 0	rt = $s2	rd = $s1	sa = 10	f = 0
srl $s1,$s2,10	$s1 = $s2>>>10	op = 0	rs = 0	rt = $s2	rd = $s1	sa = 10	f = 2
sra $s1, $s2, 10	$s1 = $s2 >> 10	op = 0	rs = 0	rt = $s2	rd = $s1	sa = 10	f = 3
sllv $s1,$s2,$s3	$s1 = $s2 << $s3	op = 0	rs = $s3	rt = $s2	rd = $s1	sa = 0	f = 4
srlv $s1,$s2,$s3	$s1 = $s2>>>$s3	op = 0	rs = $s3	rt = $s2	rd = $s1	sa = 0	f = 6
srav $s1,$s2,$s3	$s1 = $s2 >> $s3	op = 0	rs = $s3	rt = $s2	rd = $s1	sa = 0	f = 7

Instruction	Meaning	I-Type Format			
addi $s1, $s2, 10	$s1 = $s2 + 10	op = 0x8	rs = $s2	rt = $s1	imm^{16} = 10
addiu $s1, $s2, 10	$s1 = $s2 + 10	op = 0x9	rs = $s2	rt = $s1	imm^{16} = 10
andi $s1, $s2, 10	$s1 = $s2 & 10	op = 0xc	rs = $s2	rt = $s1	imm^{16} = 10
ori $s1, $s2, 10	$s1 = $s2 | 10	op = 0xd	rs = $s2	rt = $s1	imm^{16} = 10
xori $s1, $s2, 10	$s1 = $s2 ^ 10	op = 0xe	rs = $s2	rt = $s1	imm^{16} = 10
lui $s1, 10	$s1 = 10 << 16	op = 0xf	0	rt = $s1	imm^{16} = 10

Instruction	Meaning	Format			
j label	jump to label	op^6 = 2	imm^{26}		
beq rs, rt, label	branch if (rs == rt)	op^6 = 4	rs^5	rt^5	imm^{16}
bne rs, rt, label	branch if (rs != rt)	op^6 = 5	rs^5	rt^5	imm^{16}
blez rs, label	branch if (rs<=0)	op^6 = 6	rs^5	0	imm^{16}
bgtz rs, label	branch if (rs > 0)	op^6 = 7	rs^5	0	imm^{16}
bltz rs, label	branch if (rs < 0)	op^6 = 1	rs^5	0	imm^{16}
bgez rs, label	branch if (rs>=0)	op^6 = 1	rs^5	1	imm^{16}

Instruction	Meaning	Format					
slt rd, rs, rt	rd=(rs<rt?1:0)	op^6 = 0	rs^5	rt^5	rd^5	0	0x2a
sltu rd, rs, rt	rd=(rs<rt?1:0)	op^6 = 0	rs^5	rt^5	rd^5	0	0x2b
slti rt, rs, imm^{16}	rt=(rs<imm?1:0)	0xa	rs^5	rt^5	imm^{16}		
sltiu rt, rs, imm^{16}	rt=(rs<imm?1:0)	0xb	rs^5	rt^5	imm^{16}		

Instruction	Meaning	I-Type Format			
lb rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x20	rs^5	rt^5	imm^{16}
lh rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x21	rs^5	rt^5	imm^{16}
lw rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x23	rs^5	rt^5	imm^{16}
lbu rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x24	rs^5	rt^5	imm^{16}
lhu rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x25	rs^5	rt^5	imm^{16}
sb rt, imm^{16}(rs)	MEM[rs+imm^{16}] = rt	0x28	rs^5	rt^5	imm^{16}
sh rt, imm^{16}(rs)	MEM[rs+imm^{16}] = rt	0x29	rs^5	rt^5	imm^{16}
sw rt, imm^{16}(rs)	MEM[rs+imm^{16}] = rt	0x2b	rs^5	rt^5	imm^{16}

Instruction	Meaning	Format					
jal label	$31=PC+4, jump	op^6 = 3	imm^{26}				
jr Rs	PC = Rs	op^6 = 0	rs^5	0	0	0	8
jalr Rd, Rs	Rd=PC+4, PC=Rs	op^6 = 0	rs^5	0	rd^5	0	9

Instruction	Meaning	Format					
mult Rs, Rt	Hi, Lo = Rs × Rt	op^6 = 0	Rs^5	Rt^5	0	0	0x18
multu Rs, Rt	Hi, Lo = Rs × Rt	op^6 = 0	Rs^5	Rt^5	0	0	0x19
mul Rd, Rs, Rt	Rd = Rs × Rt	0x1c	Rs^5	Rt^5	Rd^5	0	0x02
div Rs, Rt	Hi, Lo = Rs / Rt	op^6 = 0	Rs^5	Rt^5	0	0	0x1a
divu Rs, Rt	Hi, Lo = Rs / Rt	op^6 = 0	Rs^5	Rt^5	0	0	0x1b
mfhi Rd	Rd = Hi	op^6 = 0	0	0	Rd^5	0	0x10
mflo Rd	Rd = Lo	op^6 = 0	0	0	Rd^5	0	0x12