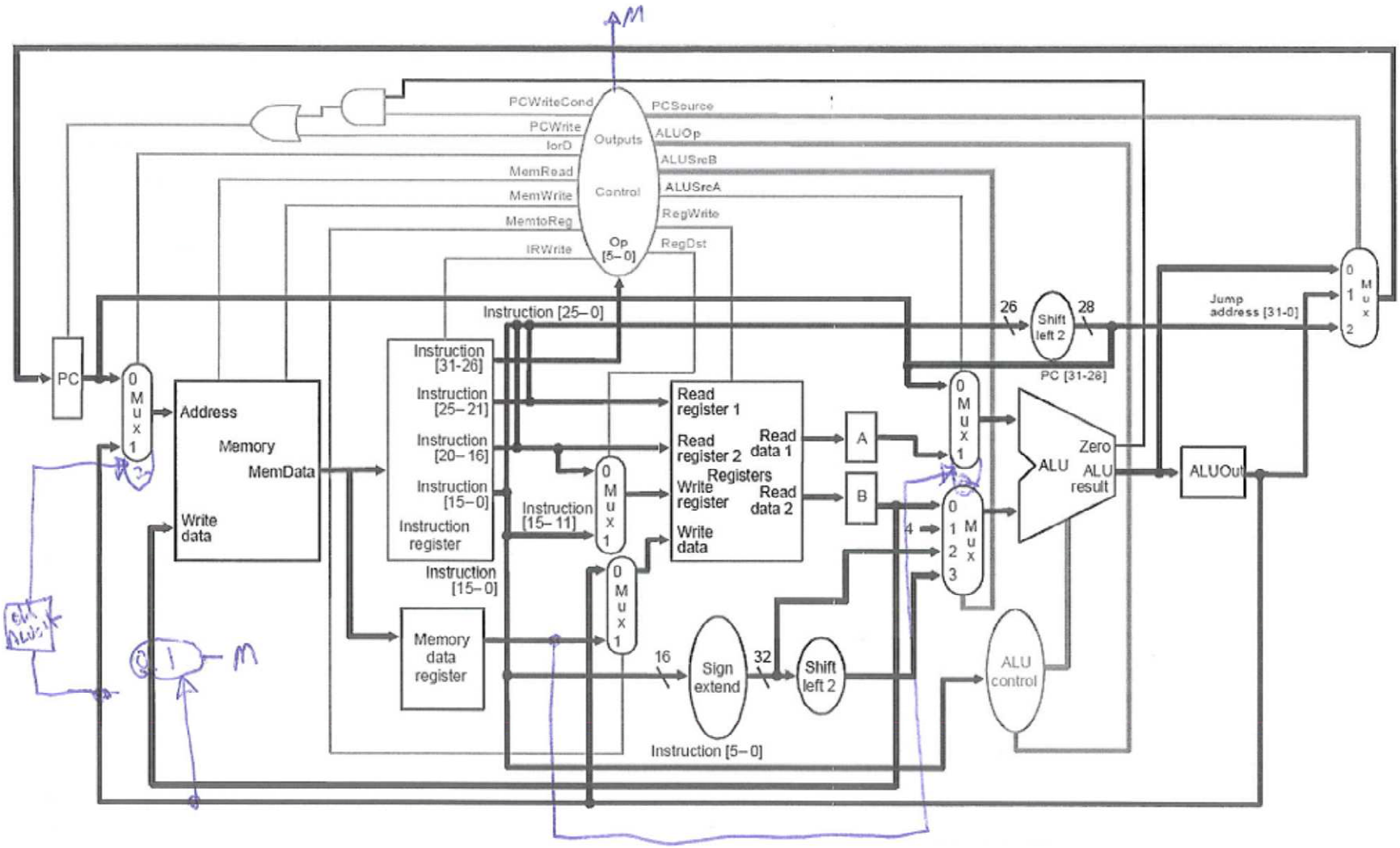
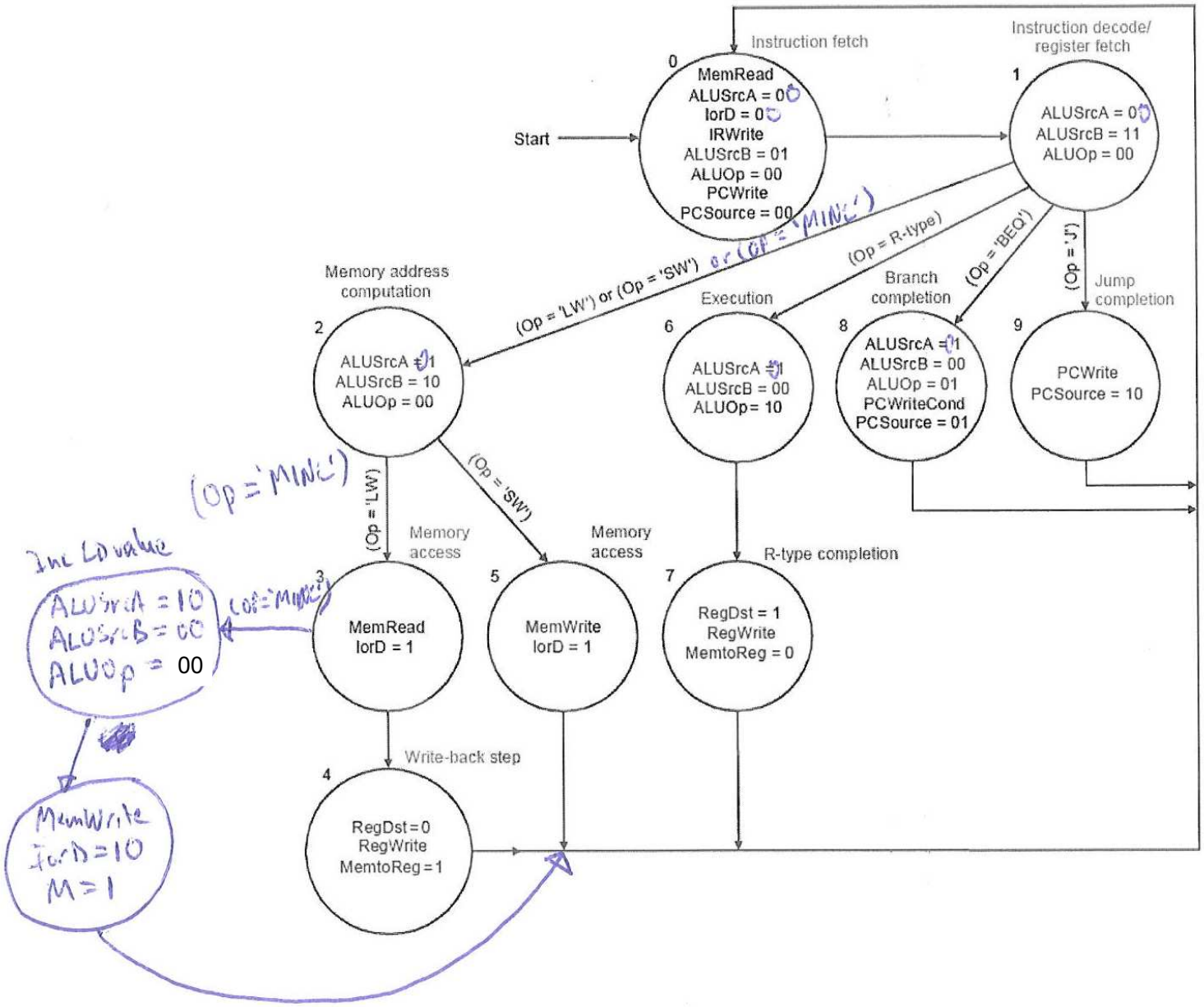
More Practice Problems

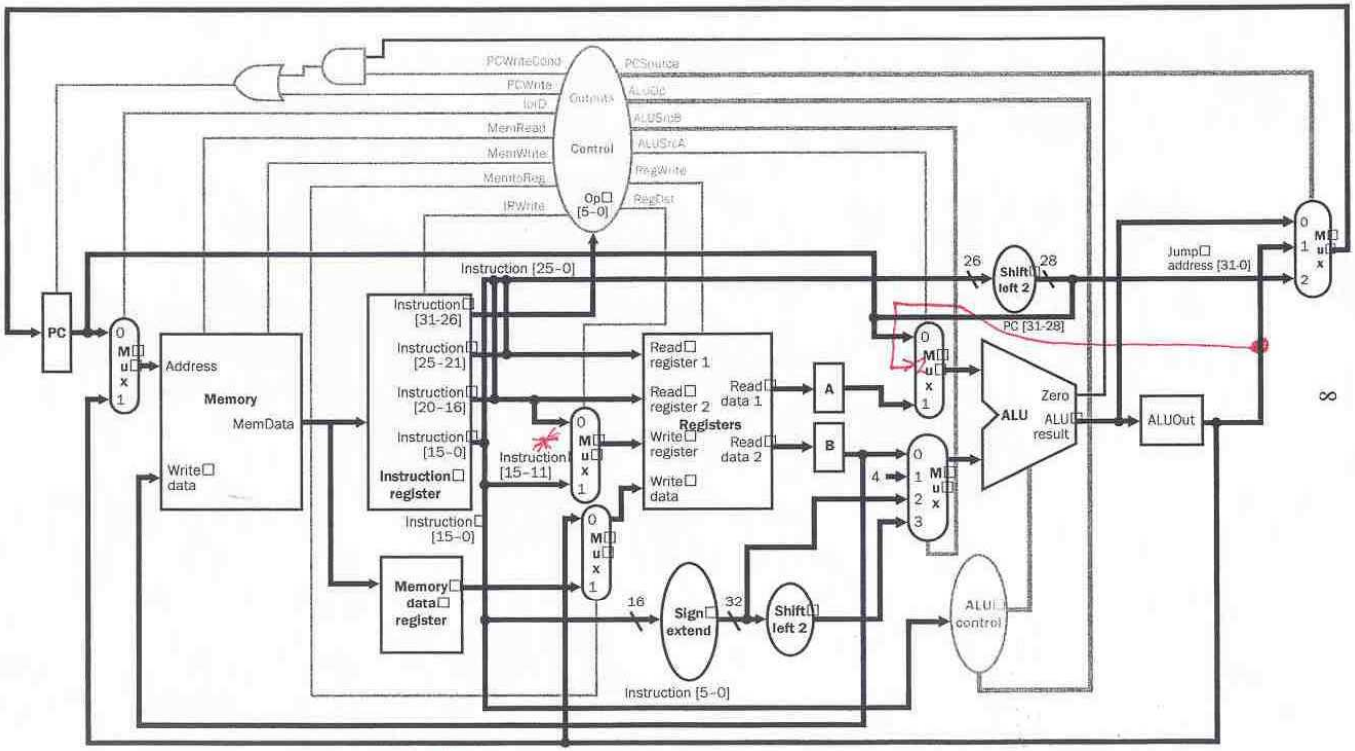
1/28/10

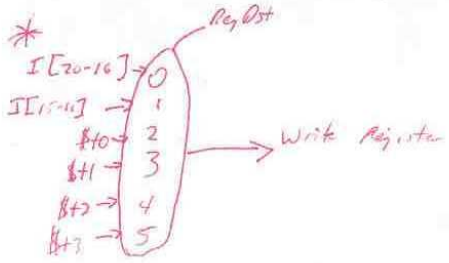
- ***1. Memory Increment (20 points)***: For this problem, extend the multicycle datapath with the memory increment instruction (*minc)*. The *minc* instruction increments the value stored at a location in memory by a value stored in a register. The *minc* instruction is an i-type instruction that uses base+offset addressing. In this instruction, the *rs* field of the i-type instruction corresponds to the register containing the base address of the memory reference, the *rt* field of the i-type instruction corresponds to the register containing the increment to be added to memory, and the immediate field contains the offset to the base address of the memory reference. We want to add the register defined by *rt* to the contents of memory at the location specified by the sum of the contents of the register defined by *rs* and the immediate field. This memory addressing is identical to what is already done for the load and store instructions in MIPS.
- Implement this instruction on the multicycle datapath and show the changes to the datapath and control.
- You must not impact other instructions when you modify the datapath or control! All other instructions should be able to execute normally on this modified datapath. The values in the registers specified by *rs* and *rt* cannot be overwritten, nor can you write to an arbitrary register location in the register file.

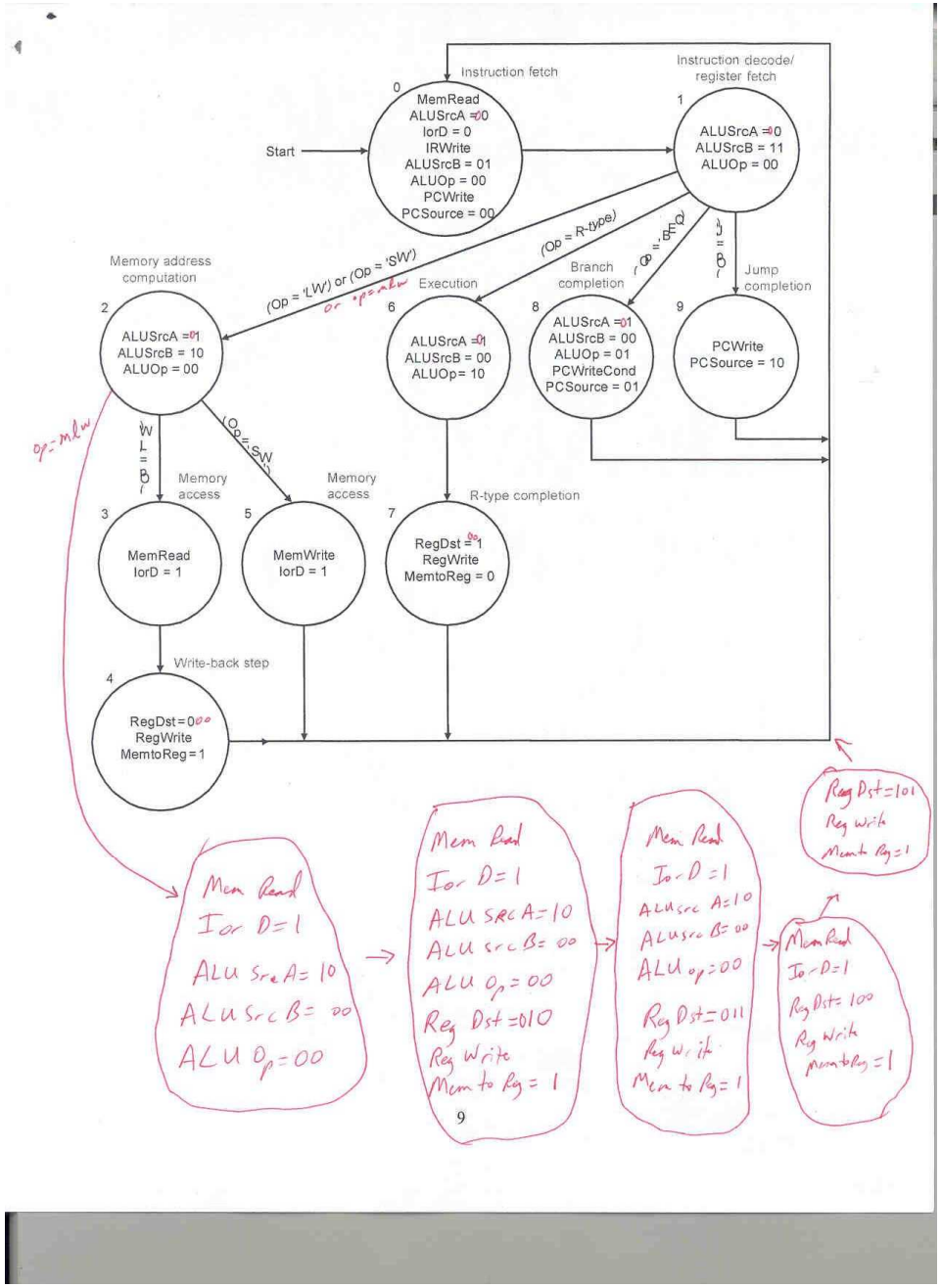

PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[31-26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction [25–0]
Instruction
[15–11]
Instruction
[15–0]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
16
Sign
extend
32
Shift
left 2
ALU
Zero
ALU
result
ALUOut
ALU
control
26
Shift
left 2
28
PC [31-28]
Jump
address [31-0]
Instruction [5–0]


Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst = 0
RegWrite
MemtoReg = 1

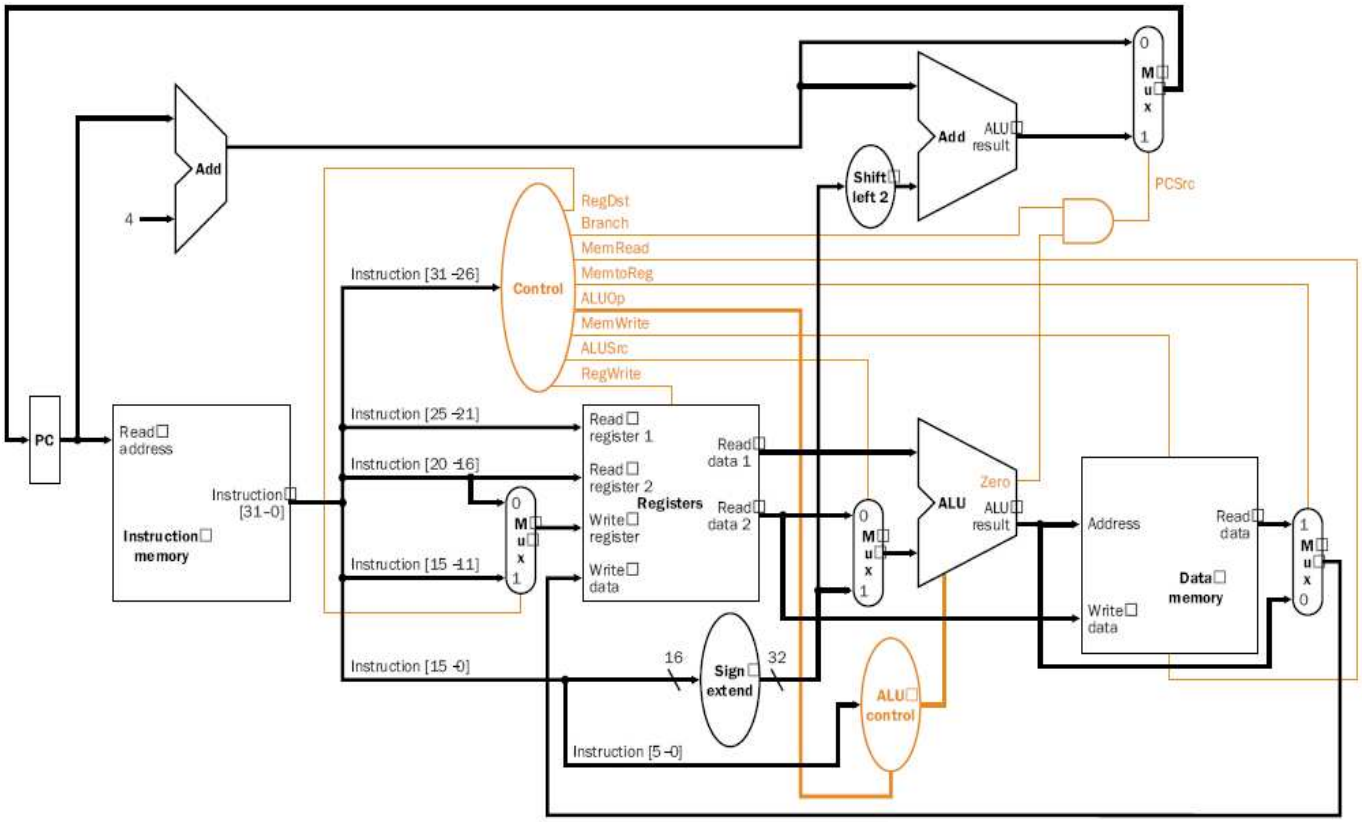
- ***2. Multiload (20 points)***: Consider an application that loads many addresses that are a fixed stride away from each other at the same time from memory. One way to reduce the instruction count in such an application would be to add a load instruction that can read a number of memory locations at once and write to a given set of registers. We will introduce the multiload (*mlw*) instruction to do just that. Our multiload instruction will read **4** words from 4 different memory locations. The multiload instruction will **always** write to registers $t0, $t1, $t2, and $t3 – no matter what.
- The *mlw* instruction uses the I-type format. As in the regular *lw* instruction, rs corresponds to the register containing the base address to load and the immediate field contains the offset to add to the base register. But the rt field for the *mlw* instruction specifies the stride through memory that we will use in obtaining our 4 memory values. Therefore the instruction *mlw* $s2, 20($s3) would have the following effect:
- $t0 = M[$s3+20]
- $t1 = M[$s3+20+$s2]
- $t2 = M[$s3+20+$s2+$s2]
- $t3 = M[$s3+20+$s2+$s2+$s2]
- Note that *mlw* is still one instruction, but it writes multiple registers and reads multiple memory locations – and further note that the address loaded changes for each access by adding rt.
- Implement this instruction on the multicycle datapath and show the changes to the datapath and control.


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Outputs
Control
Op [5-0]
PC
Address
Memory
MemData
Write data
Instruction [31-26]
Instruction [25-21]
Instruction [20-16]
Instruction [15-0]
Instruction register
Instruction [15-0]
Instruction [15-11]
Memory data register
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
A
B
4
ALU
Zero
ALU result
ALUOut
26
Shift left 2
28
PC [31-28]
Jump address [31-0]
16
Sign extend
32
Shift left 2
ALU control
Instruction [5-0]
Instruction [25-0]


RegDst
I[20-16]
I[15-11]
$t0
$t1
$t2
$t3
0
1
2
3
4
5
Write Register


Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'JMP')
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
3
MemRead
IorD = 1
Memory
access
5
MemWrite
IorD = 1
R-type completion
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst = 0
RegWrite
MemtoReg = 1
op = lw
Mem Read
Ior D = 1
ALU Src A = 10
ALU Src B = 00
ALU Op = 00
Mem Read
Ior D = 1
ALU src A = 10
ALU src B = 00
ALU op = 00
Reg Dst = 010
Reg Write
Mem to Reg = 1
Mem Read
Ior D = 1
ALUsrc A = 10
ALUsrc B = 00
ALU op = 00
Reg Dst = 011
Reg Write
Mem to Reg = 1
Mem Read
Ior D = 1
Reg Dst = 100
Reg Write
memtoReg = 1
Reg Dst = 101
Reg Write
Mem to Reg = 1
9

- ***3. Don't Get Jalm-ed Up On This One (20 points)***: Consider the *jalm* instruction – it jumps to a new location based on a register value after storing the current PC+4 to a location in memory. In summary, it is an I-type instruction that has two effects:
  - Writes PC+4 to the memory location at the base+displacement address using rs and the immediate field
- i.e. M[R[rs]+SE(I)] = PC+4
  - Changes the PC to the value stored in the register specified by rt
- i.e. PC = R[rt]
- So for example,
- jalm $t1, 4 ($t2)
- would have the following effects:
- 1) Memory[$t2+4] = PC + 4
- 2) PC = $t1
- Implement *jalm* on the **single cycle datapath**. Use the I-type instruction format. Implement your solution on the following two pages.
- All other instructions must still work correctly after your modifications. You should **not** add any new ALUs, register file ports, or ports to memory.

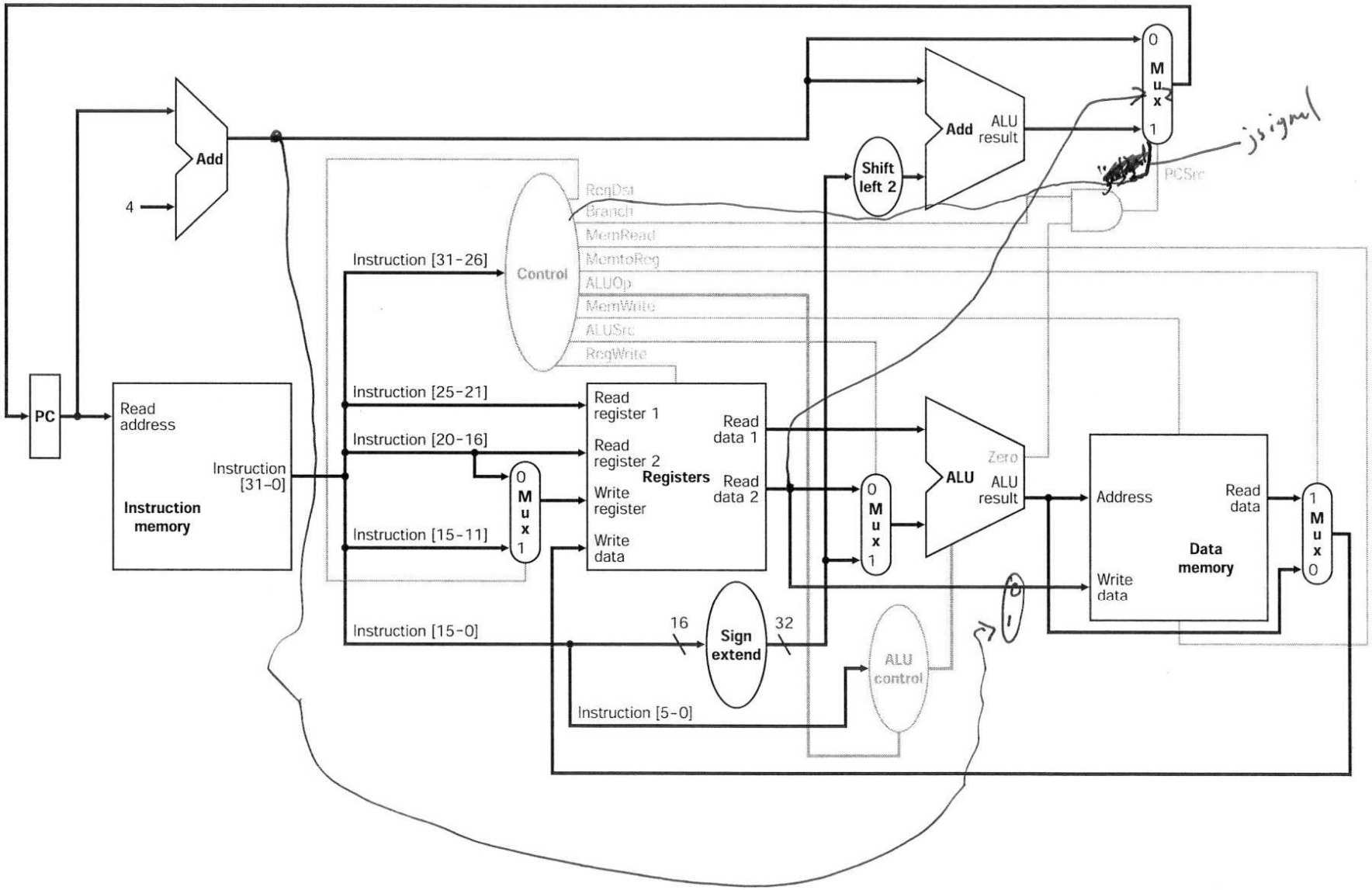


ALU Controller

Inputs		ALU Action	Output
ALUOp	function		ALUCtrl
00	XXXXXX	Add	010
00	XXXXXX	Add	010
01	XXXXXX	Subtract	110
10	100000	Add	010
10	100010	Subtract	110
10	100100	AND	000
10	100101	OR	001
10	101010	SLT	111

Main Controller

Input or Output	Signal Name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1


Add
4
PC
Read address
Instruction [31-0]
Instruction memory
Instruction [31-26]
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [25-21]
Instruction [20-16]
Instruction [15-11]
Instruction [15-0]
Instruction [5-0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
0 M u x 1
16
Sign extend
32
Shift left 2
Add
ALU result
0 M u x 1
PCSrc
ALU
Zero
ALU result
ALU control
Address
Read data
Data memory
Write data
1 M u x 0

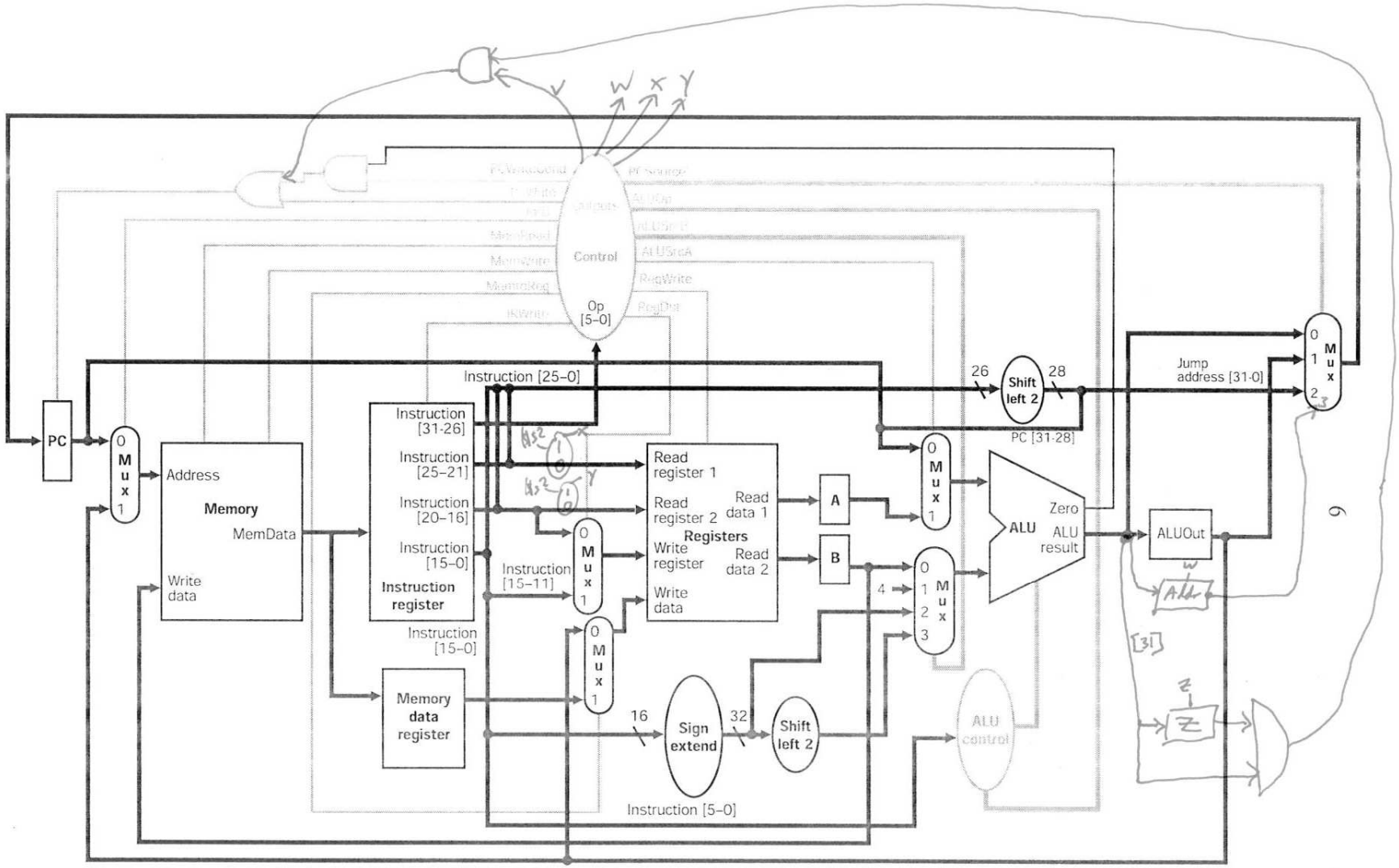
Main Controller

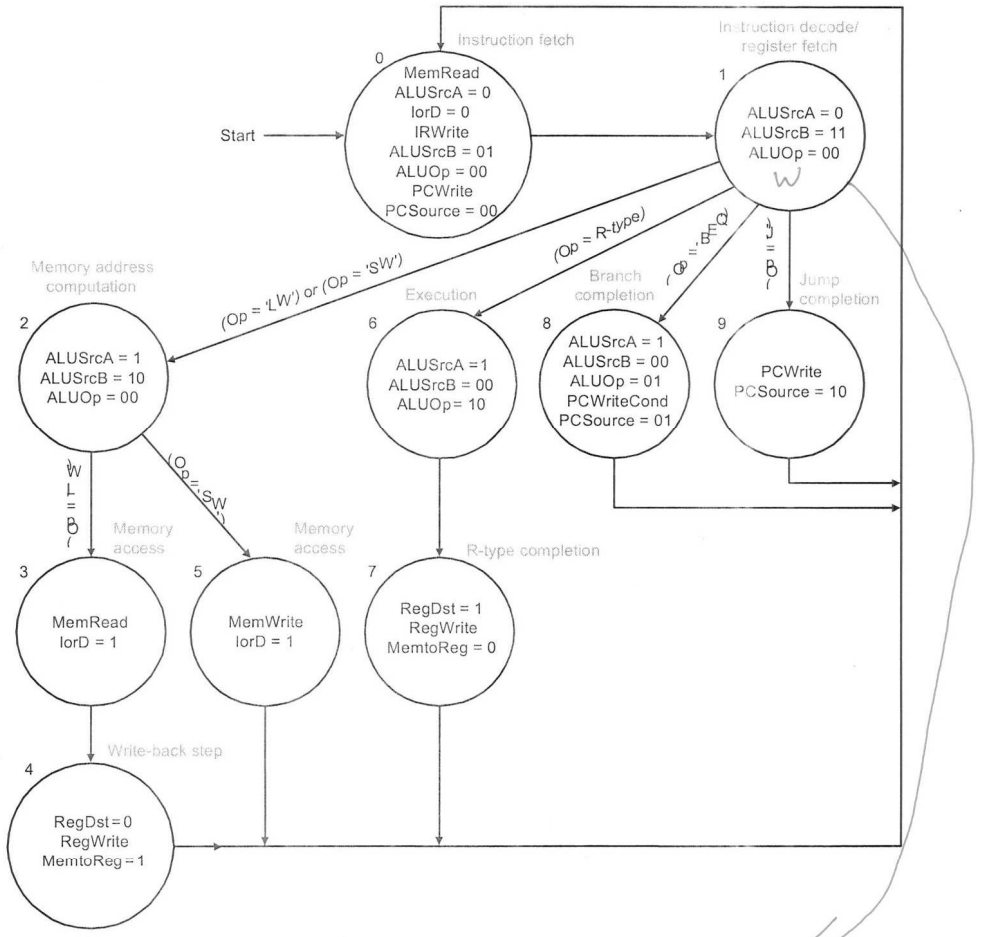
Input or Output	Signal Name	R-format	lw	sw	Beq	Jaln
Inputs	Op5	0	1	1	0	
	Op4	0	0	0	0	
	Op3	0	0	1	0	
	Op2	0	0	0	1	
	Op1	0	1	1	0	
	Op0	0	1	1	0	
Outputs	RegDst	1	0	X	X	X
	ALUSrc	0	1	1	0	1
	MemtoReg	0	1	X	X	X
	RegWrite	1	1	0	0	0
	MemRead	0	1	0	0	0
	MemWrite	0	0	1	0	1
	Branch	0	0	0	1	0
	ALUOp1	1	0	0	0	0
	ALUOp0	0	0	0	1	0
	Jsignal	0	0	0	0	1

ALU Controller

opcode	**ALUOp**	**instruction**	**function**	**ALU Action**	**ALUCtrl**
lw	00	load word	XXXXXX	add	010
sw	00	store word	XXXXXX	add	010
beq	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	AND	000
R-type	10	OR	100101	OR	001
R-type	10	SLT	101010	SLT	111

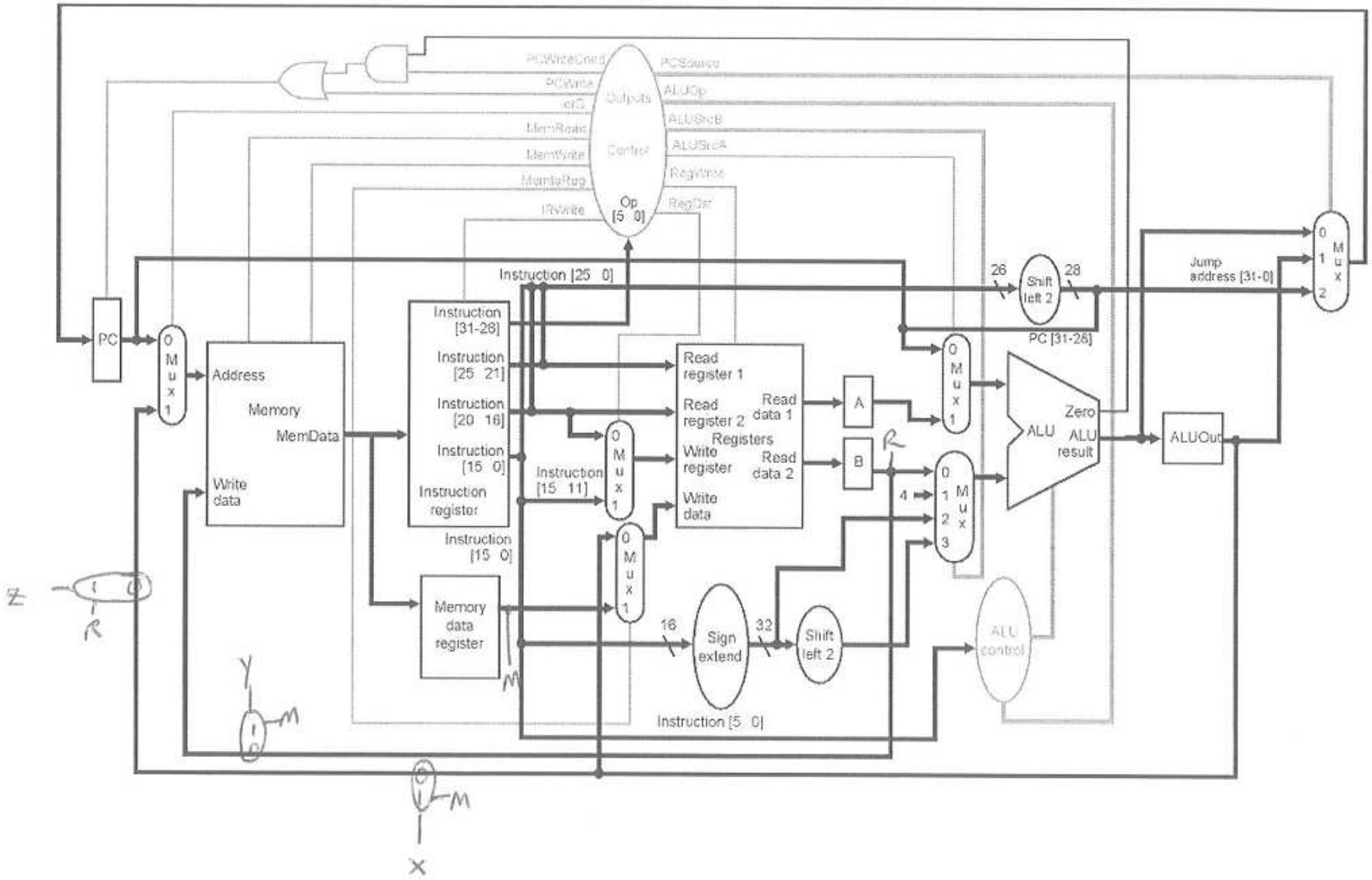
- ***4. Battle of the BLG (38 points)***: You find that the following sequence of code is frequently executed on the multicycle datapath for the applications you are considering:
- *slt $t4, $s2, $s1*
- *slt $t5, $s3, $s2*
- *and $t4, $t4, $t5*
- *bne $t4, $zero, Label*
- Basically, this code is checking to see if a particular value (in the above case this is stored in $s2) falls between two other values (in the above case, these are stored in $s1 and $s3). Furthermore, the intermediate values (in the above case, $t4 and $t5) are only used by the above instructions and do not need to be kept live after these four instructions execute (i.e. no other instructions use these values after the *bne*). You also observe that the value being compared is **always** stored in $s2. Finally, the *bne* instruction in the sequence you observe always compares against the value 0.
- You see an opportunity to replace this sequence of instructions with a single I-type instruction – the ***branch if less-than and greater-than (blg)*** instruction. This instruction will have the following behavior:
- if (R[rs] < R[$s2] < R[rt])
- PC = PC + 4 + SignExtend(Immediate<<2)
- Note again that this is always comparing against $r2, but that rs and rt will depend on the instruction. Furthermore, you should **not** write to any compiler visible registers except the PC – part of the benefit of this instruction is that it uses fewer registers than the above code sequence.
  - Use the multicycle datapath to implement the instruction.
- Implement this instruction on the following two pages. All other instructions must still work correctly after your modifications. You should **not** add any new ALUs, register file ports, or ports to memory.

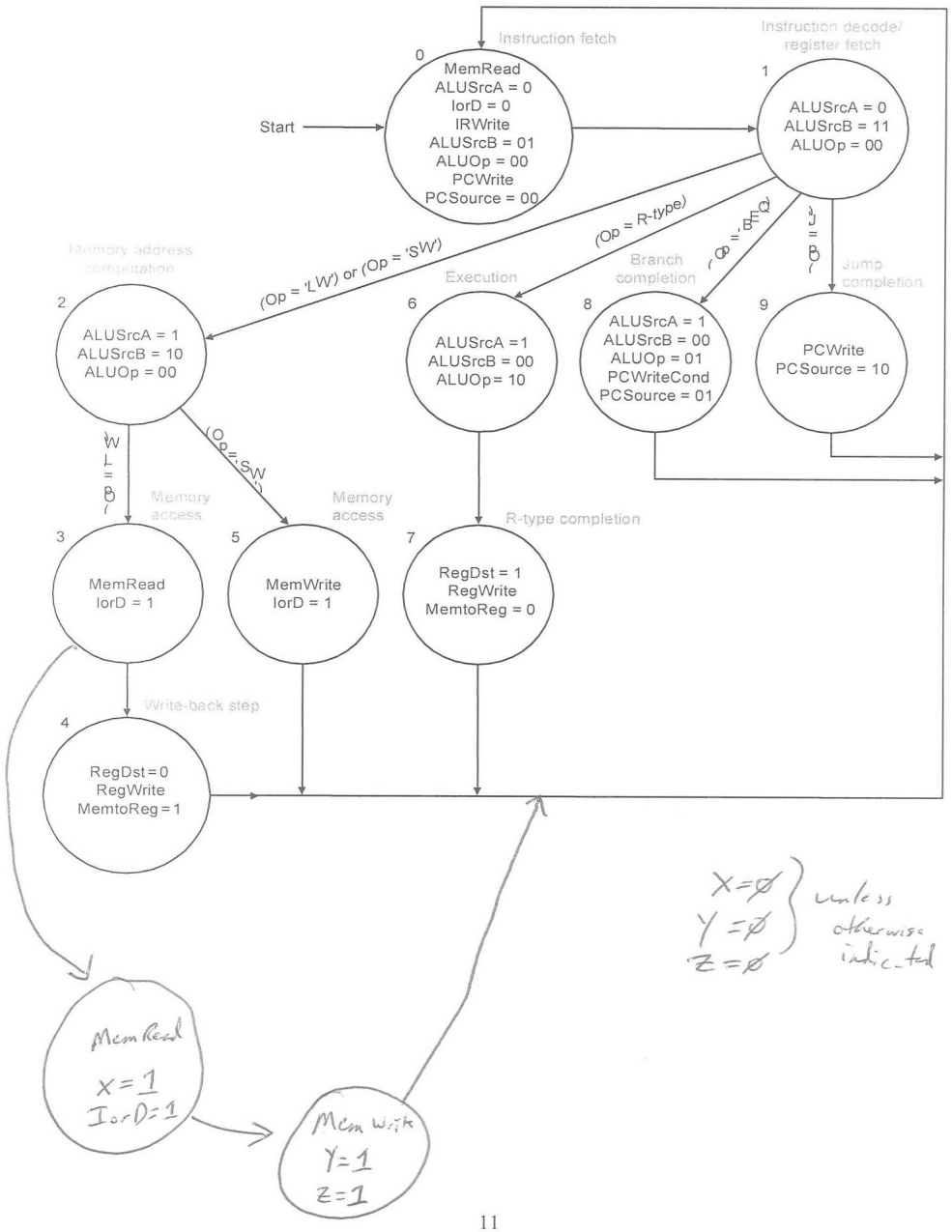

Control
Op
[5–0]
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[31-26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction [25–0]
Instruction
[15–11]
Instruction
[15–0]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
16
Sign
extend
32
Shift
left 2
26
Shift
left 2
28
PC [31-28]
Jump
address [31-0]
ALU
Zero
ALU
result
ALUOut
ALU
control
Instruction [5–0]



Except for the * states, Y=0 and X=0

- ***5. A Little Indirection (20 points)***: Use the multicycle datapath to implement the ***memory indirect store (mis)*** instruction. This instruction uses the I-type instruction format, and has the following behavior:
- M[R[rt]] = M[M[R[rs] + SignExtend(Immediate)]]
-
- Note very carefully the placement of []'s in this description!
- Implement this instruction on the following two pages. All other instructions must still work correctly after your modifications. You should **not** add any new ALUs, register file ports, or ports to memory.


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5 0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[31-28]
Instruction
[25 21]
Instruction
[20 16]
Instruction
[15 0]
Instruction
register
Instruction [25 0]
Instruction
[15 11]
Instruction
[15 0]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
ALU
Zero
ALU
result
ALUOut
26
Shift
left 2
28
PC [31-28]
Jump
address [31-0]
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5 0]
Z
R
Y
M
X


Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = BEQ)
(Op = JMP)
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA =1
ALUSrcB = 00
ALUOp= 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = LW)
(Op = SW)
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst = 0
RegWrite
MemtoReg = 1
X = ∅
Y = ∅
Z = ∅
unless
otherwise
indicated
MemRead
X = 1
IorD = 1
MemWrite
Y = 1
Z = 1