CSE 2021
COMPUTER ORGANIZATION

HUGH CHESSER
CSE B 1012U

W9-M

Agenda

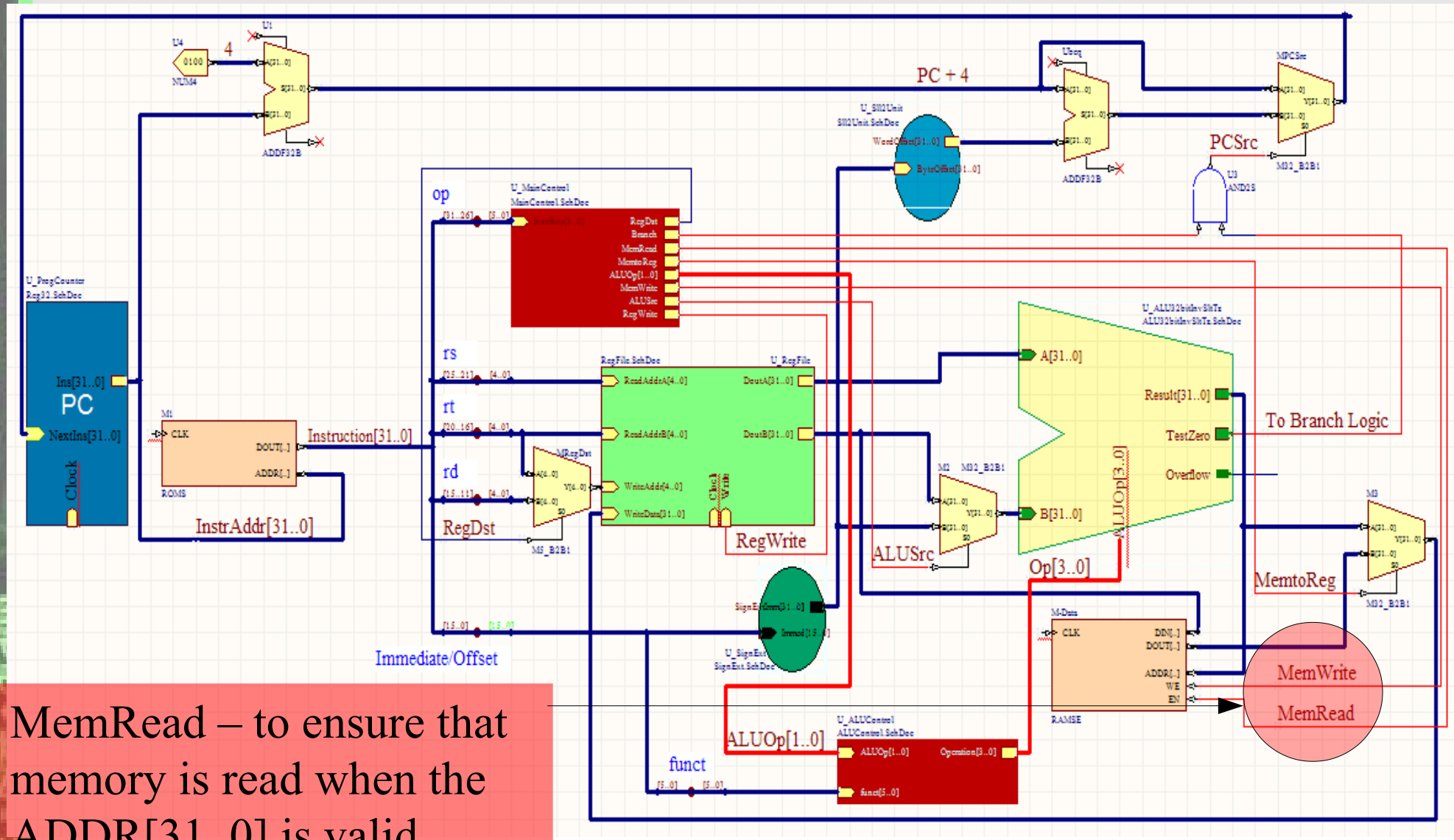
Topics:

1. Sample Exam/Quiz Q - Review
2. Multiple cycle implementation

Patterson: Section 4.5

Reminder: Quiz #2 – Next Wednesday

Main Control (4)



MemRead – to ensure that memory is read when the ADDR[31..0] is valid

Activity (Sample Quiz, Exam Q)

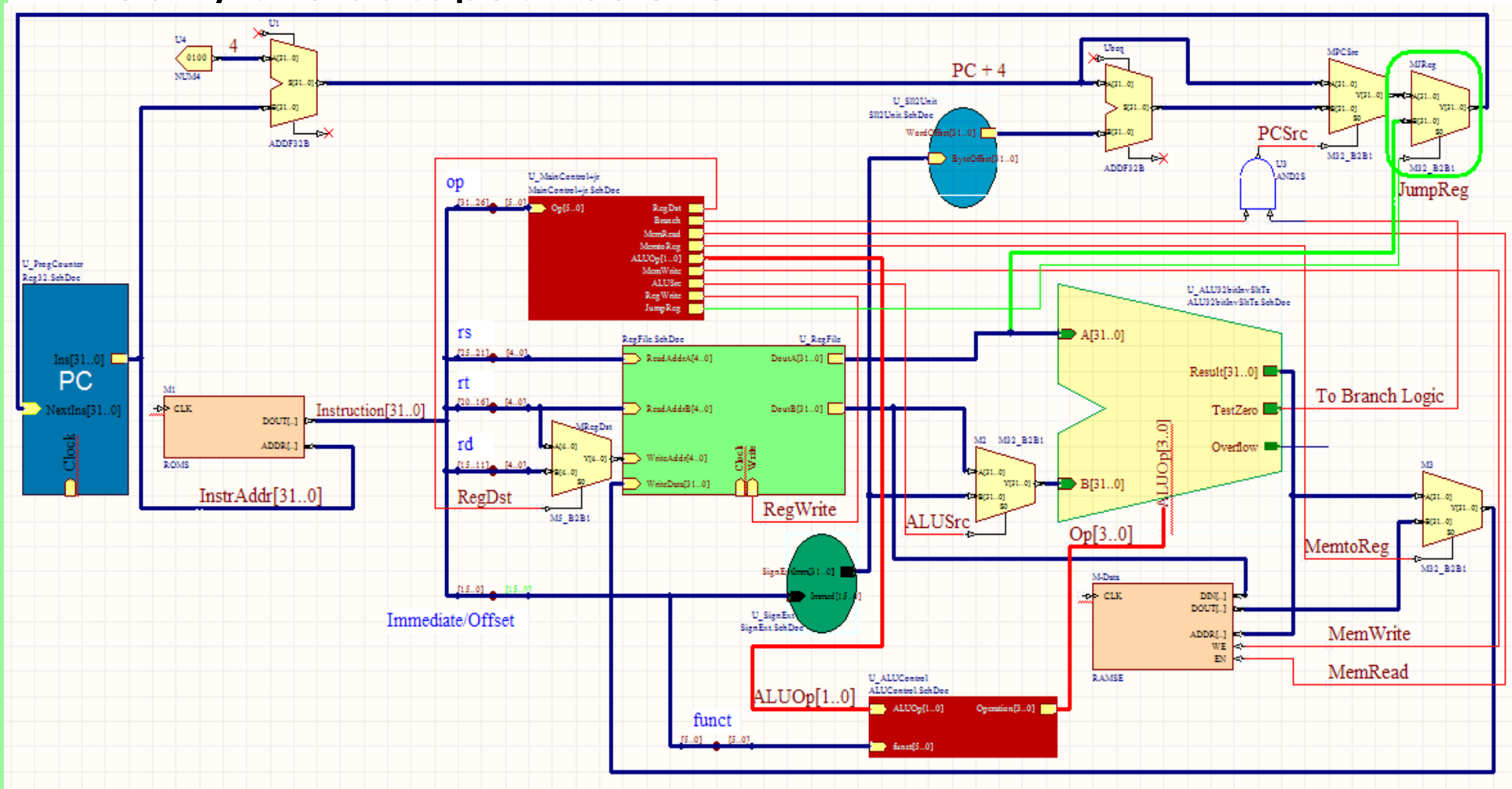
We wish to add jr (jump register) to the single cycle datapath from the previous slide. Add the necessary connections to the single cycle datapath block diagram to implement the jr instruction. Also, append the table below to add the necessary control signals needed for the jr instruction.

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**
`lw`	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**
`sw`	**X**	**1**	**X**	**0**	**0**	**1**	**0**	**0**	**0**
`beq`	**X**	**0**	**X**	**0**	**0**	**0**	**1**	**0**	**1**

Answer (Part 1):

Jump Register	jr	R	PC=R[rs]	$0 / 08_{hex}$

Modify the datapath as shown



Answer (Part 2):

… append the table below to add the necessary control signals needed for the jr instruction.

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	JumpReg	ALUOp1	ALUOp0
R-format	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**
`lw`	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**
`sw`	**X**	**1**	**X**	**0**	**0**	**1**	**0**	**0**	**0**	**0**
`beq`	**X**	**0**	**X**	**0**	**0**	**0**	**1**	**0**	**0**	**1**
`jr`	**X**	**0**	**X**	**0**	**0**	**0**	**0**	**1**	**X**	**X**

Jump Register	`jr`	R	PC=R[rs]	$0 / 08_{hex}$

Why Multicycle?

Example: Assume that the operation times for major functional unit in a microprocessor are:

Memory unit ~ 2ns, ALU and adders ~ 2ns, Register file ~ 1ns

Compare the performance of the following instruction mix

Loads: 24%; Stores: 12%; ALU instructions: 44%; Branches: 18%; Jumps: 2%

on the two implementations

Implementation I: All instructions operate in 1 clock cycle

Implementation II: Each instruction is as long as it needs to be.

Instruction Class	Functional units used (Steps involved)					
ALU type	**Instruction fetch**	**Register Access**	**ALU**	**Register Access**		**6ns**
Load word	**Instruction fetch**	**Register Access**	**ALU**	**Memory Access**	**Register Access**	**8ns**
Store word	**Instruction fetch**	**Register Access**	**ALU**	**Memory Access**		**7ns**
Branch	**Instruction fetch**	**Register Access**	**ALU**			**5ns**
Branch	**Instruction fetch**					**2ns**

Average time per instruction: Implementation 1: ~ 8ns

Implementation 2: ~0.24(8)+0.12(7)+0.44(6)+0.18(5)+0.02(2) = 6.34ns

Multicycle Implementation

Instruction:

- Execution of each instruction is broken into different steps
- Each step requires 1 clock cycle
- Each instruction takes multiple clock cycles

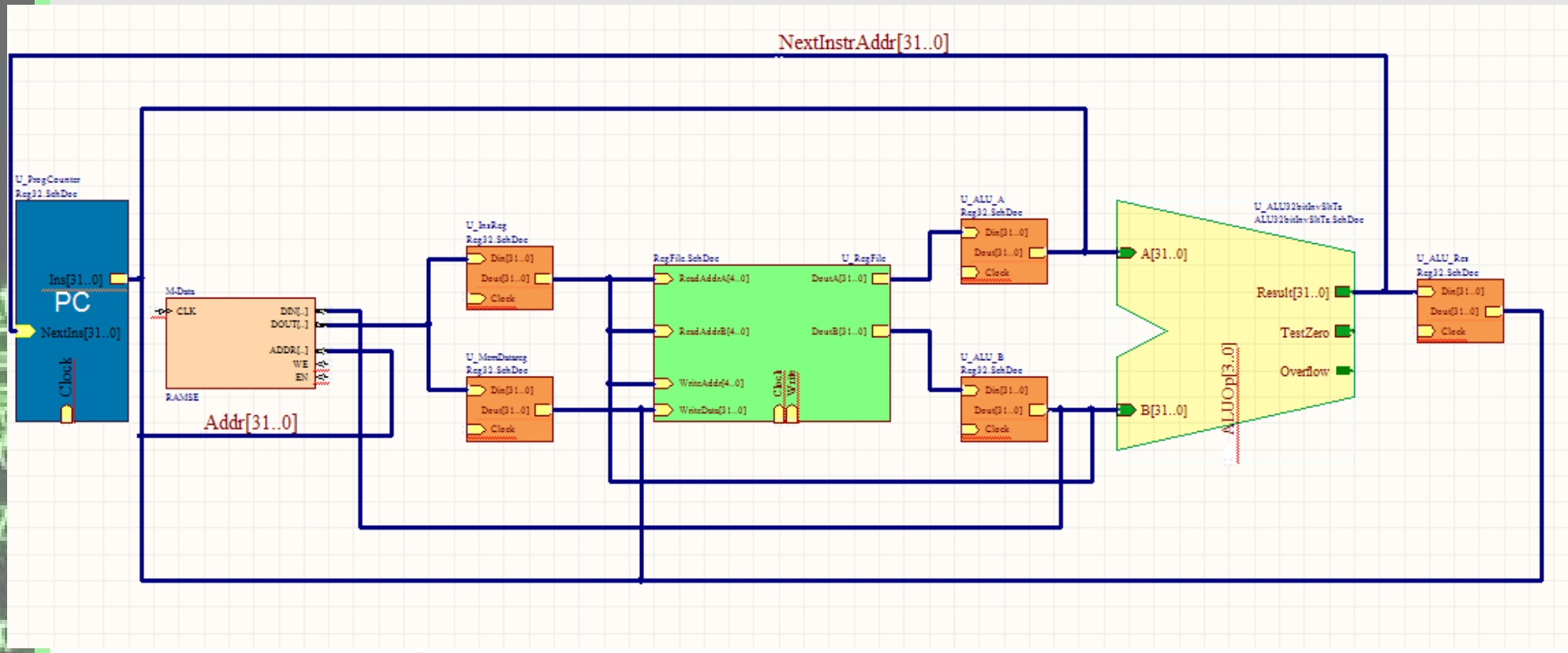
Functional Unit:

- Can be used more than once in an instruction (but still only once in a clock cycle)

Advantages:

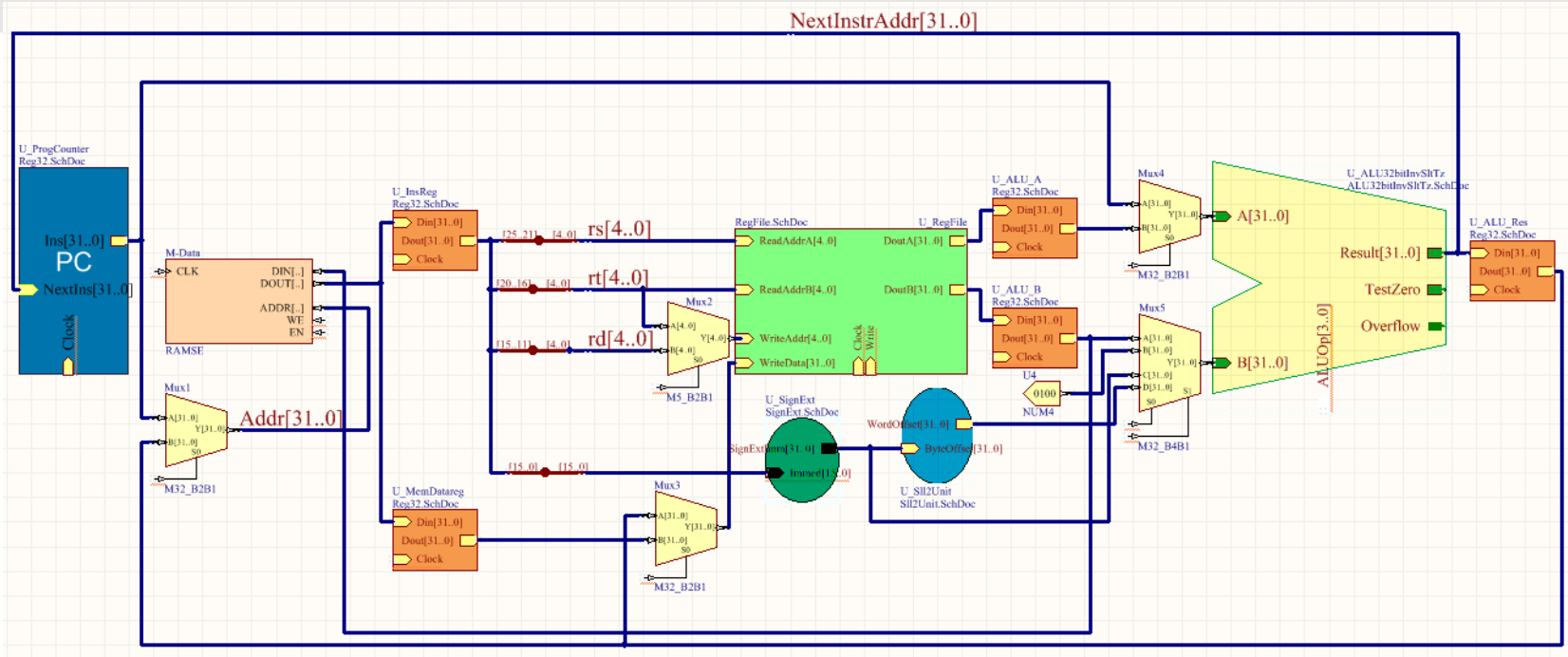
- Functional units can be shared
- ALU and adder is combined
- Single memory is used for instructions and data

Multicycle Implementation: Abstract Diagram



- Data memory and Instruction memory are combined
- 5 additional registers are added
  1. An instruction register (IR) to hold instructions before distributing data to register file or ALU
  2. A memory data register (MDR) to hold data before distributing to register file or ALU
  3. Registers A and B that hold data before the ALU
  4. Register ALUout that hold data computed by ALU

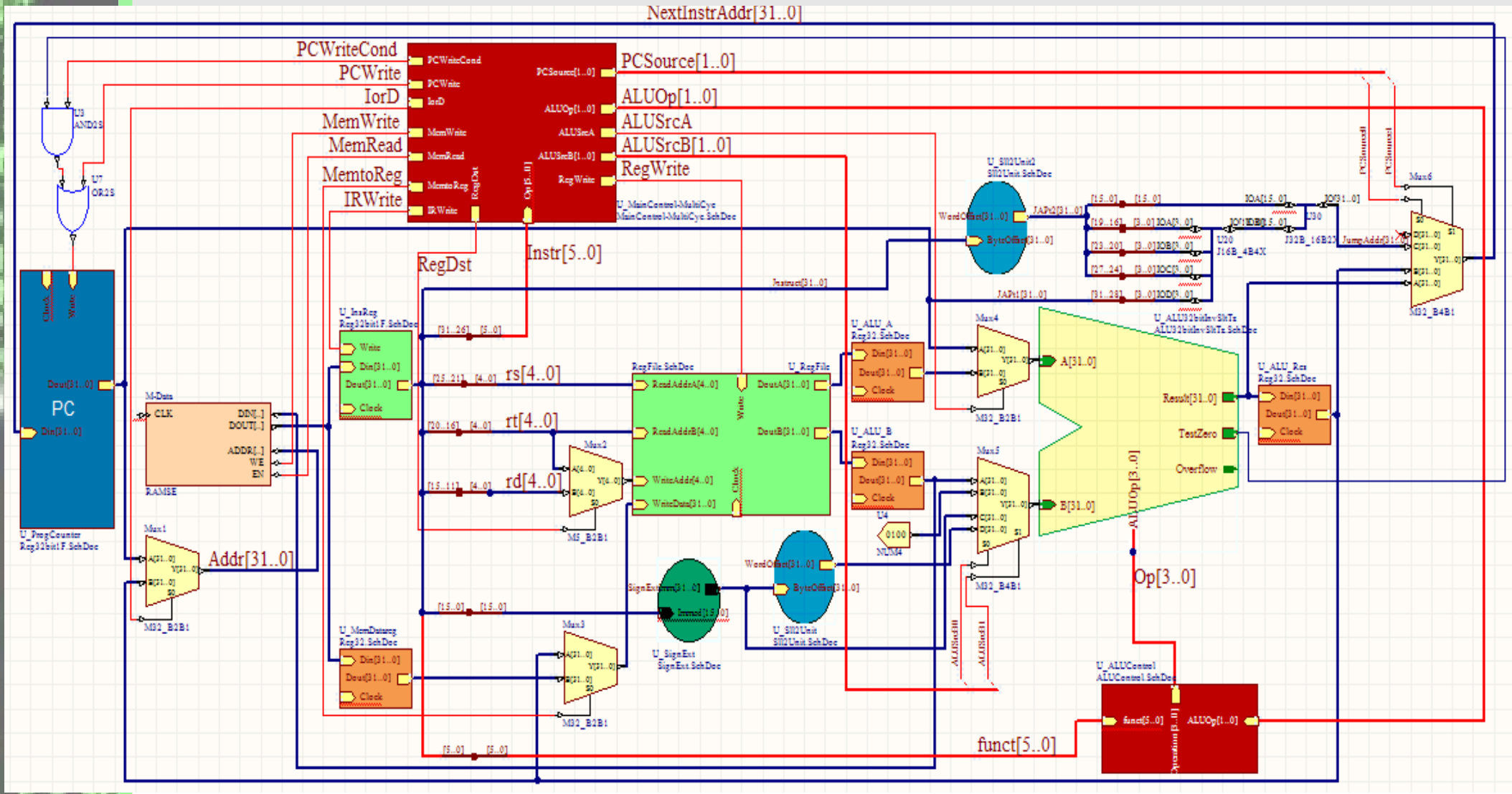
Multicycle Implementation: Multiplexers added



Because functional units are shared, multiplexers are added to select data between different devices

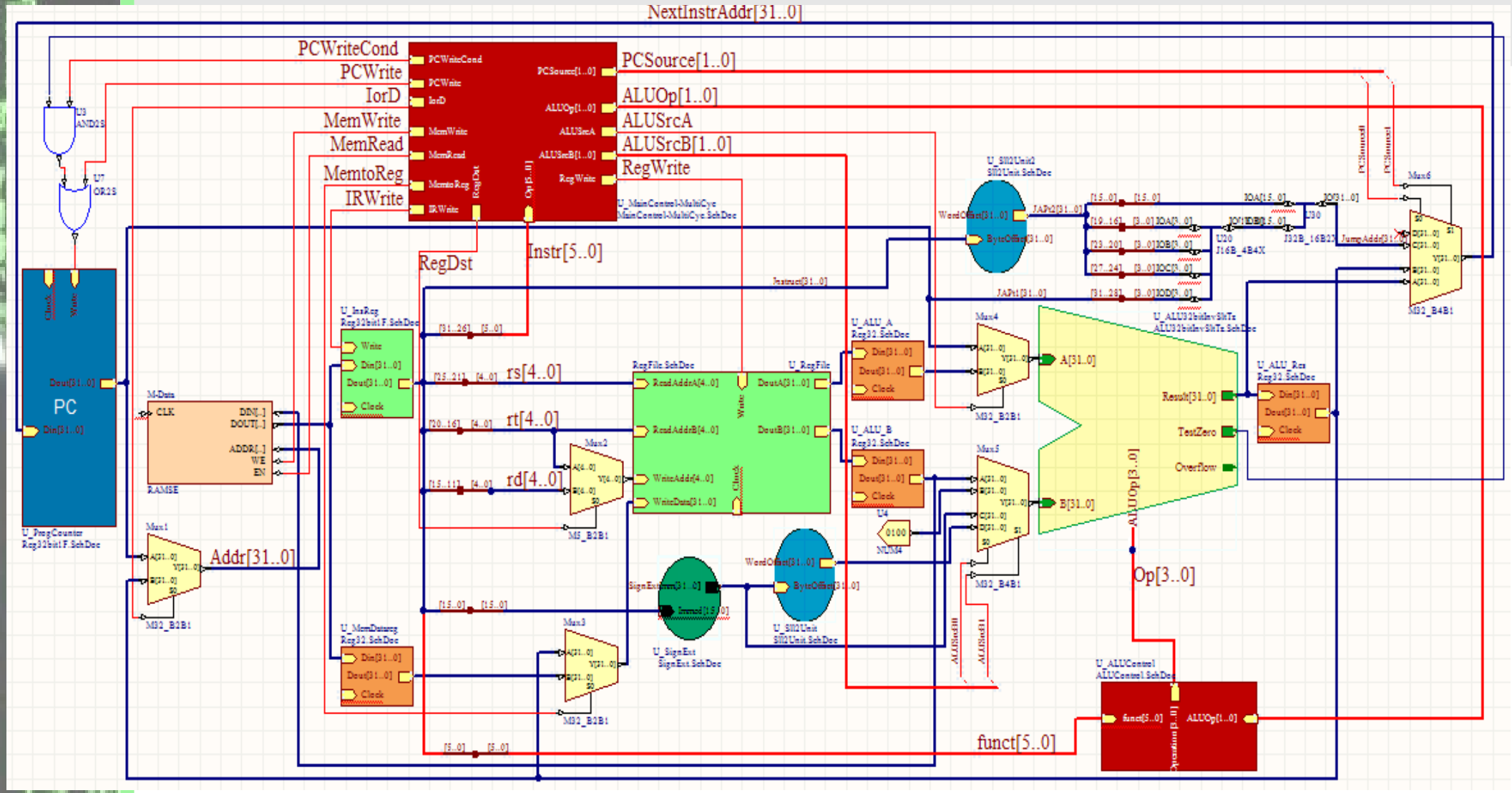
1. MUX before memory selects either the PC output (fetch instruction) or ALU output (storing data)
2. MUX before "write register" selects write-register number (instruction [15-11] or instruction[20-16])
3. MUX before "write data" selects data from "ALUOut" (R-type instruction) or "MemData" (lw instruction)
4. Upper MUX before ALU selects PC output (increment PC) or "Read data 1" (R-type instruction)
5. Lower MUX before ALU selects "Read data 2", or "sign extended instruction[15-0]" or shift left sign extended instruction[15-0], or 4

Multicycle Implementation: Controls added



1. MUX before memory selects either the PC output (fetch instruction) or ALU output (storing data)
2. MUX before "write register" selects write-register number (instruction [15-11] or instruction[20-16])

Multicycle Implementation: Control Units added



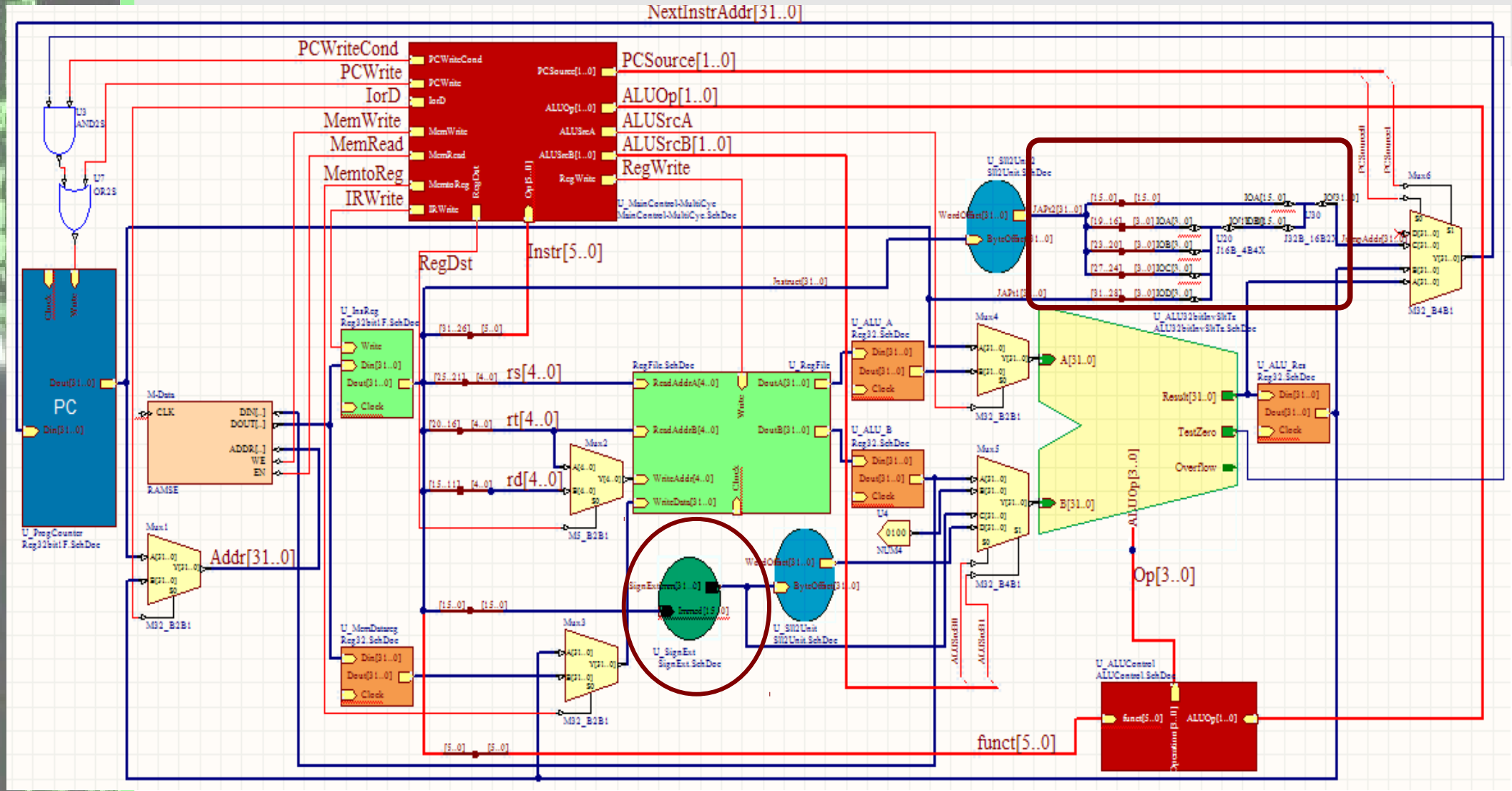
Action of 1-bit Control Signals

Control Input	Effect when Deasserted (0)	Effect when asserted (1)
IorD	PC supplies address to memory (instruction fetch)	ALUout supplies address to memory (lw/sw)
MemRead	None	Memory content specified by address is placed on "Memdata" o/p (lw/any instruction)
MemWrite	None	I/p "Write data" is stored at specified address (sw)
IRWrite	None	"MemData" o/p is written on IR (instruction fetch)
RegDst	"Write Register" specified by Instruction[20-16] (lw)	"WriteRegister" specified by Instruction[15-11] (R-type)
RegWrite	None	Data from "WriteData" i/p is written on the register specified by "WriteRegister" number
ALUSrcA	PC is the first operand in ALU (increment PC)	Register A is the first operand in ALU
MemtoReg	"WriteData" of the register file comes from ALUOut	"WriteData" of the register file comes from MDR
PCWrite	Operation at PC depends on PCWriteCond and zero output of ALU	PC is written; Source is determined by PCSource
PCWriteCond	Operation at PC depends on PCWrite	PC is written if zero o/p of ALU = 1; Source is determined by PCSource

Action of 2-bit Control Signals

Control Input	Value	Effect
ALUOp	00	ALU performs an add operation
	01	ALU performs a subtract operation
	10	The function field of Instruction defines the operation of ALU
ALUSrcB	00	The second operand of ALU comes from Register B
	01	The second operand of ALU = 4
	10	The second operand of ALU is sign extended Instruction[15-0]
	11	The second operand of ALU is sign extended, 2-bit left shifted Instruction[15-0]
PCSource	00	Output of ALU (PC + 4) is sent to PC
	01	Contents of ALUOut (branch target address = PC + 4 + 4 x offset) is sent to PC
	10	Contents of Instruction[25-0], shift left by 2, and concatenated with the MSB 4-bits of PC is sent to PC (jump instruction)

Multicycle Implementation: Control Units added



Shift Left 2?

What two instructions require the “Shift Left 2” block?

Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}

(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }

Jump	j	J	PC=JumpAddr	(5)	2_{hex}

(5) JumpAddr = { PC+4[31:28], address, 2'b0 }

Breaking the Instruction Execution into Clock Cycles

Execution of each instruction is broken into a series of steps

- Each step is balanced to do almost equal amount of work
- Each step takes one clock cycle
- Each step contains at the most 1 ALU operation, or 1 register file access, or 1 memory access
- Operations listed in 1 step occurs in parallel in 1 clock cycle
- Different steps occur in different clock cycles
- Different steps are:
  1. IF: Instruction fetch step
  2. ID: Instruction decode and register fetch step
  3. EX: Execution, memory address computation, or branch completion step
  4. MEM: Memory access of R-type instruction completion step
  5. WB: Write back completion step

Step 1: Instruction Fetch

Fetch instruction from memory and compute the address of next sequential instruction

```
IR = Memory[PC];
PC = PC + 4;
```

Operation:

1. Send PC to the memory as address (IorD = 0)
2. Read memory cell defined by PC (MemRead = 1)
3. Copy output of memory (MeMdata) into IR (IRwrite = 1)
4. Increment PC by 4 (ALUSrcA = 0, ALUSrcB = 01, PCSrc = 00)
5. Store (PC + 4) into PC (PCWrite = 1)

Step 2: Instruction Decode and Register Fetch

Read register rs in register file and store content of rs in register A

Read rt in register file and store content of rt from register file

Compute branch target address

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

Operation:

1. Access register file to write rs in A.
2. Access register file to write rt in B.
3. Compute branch target address and store in ALUOut (ALUSrcA = 0; ALUSrcB = 11)
   Remember that ALU must add (ALUOp = 00)

After this step, one of the four actions are possible: Memory reference (lw/sw), R-type, Branch, or Jump

Step 2: Instruction Decode and Register Fetch

Read register rs in register file and store content of rs in register A

Read rt in register file and store content of rt from register file

Compute branch target address

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

Operation:

1. Access register file to write rs in A.
2. Access register file to write rt in B.
3. Compute branch target address and store in ALUOut (ALUSrcA = 0; ALUSrcB = 11)
   Remember that ALU must add (ALUOp = 00)

After this step, one of the four actions are possible: Memory reference (lw/sw), R-type, Branch, or Jump

Step 3: Execution, Memory address Computation, or Branch Completion

Memory Reference (sw/lw):

```
ALUOut = A + sign-extend(IR[15-0])
```

ALU adds content of A and sign-extend(IR[15-0]) (ALUSrcA = 1, ALUSrcB = 10), (ALUOp = 00)

R-type (add/sub/or/and):

```
ALUOut = A op B
```

ALU performs specified operation on A and B (ALUSrcA = 1, ALUSrcB = 00),
Operation of ALU is determined by the function field code (ALUOp = 10)

Branch (beq):

```
if (A == B) PC = ALUOut;
```

ALU does the equal comparison operation on A and B (ALUSrcA = 1, ALUSrcB = 00),
ALU must subtract (ALUOp = 01)
Update PC with ALUOut if A == B (PCWriteCond = 1, PCSource = 01)

Jump (j):

```
PC = PC[31-28] || (IR[25-0) << 2);
```

PC gets overwritten by output of jump address MUX (PCSource = 10, PCWrite = 1)

Step 4: Memory Access or R-type Instruction Completion

Memory Reference (sw/lw):

```
MDR = Memory[ALUOut];          (for lw)
or Memory[ALUOut] = B;         (for sw)
```

1. Address from ALUOut is applied at "address" i/p of memory (IorD = 1)
2. For sw, MemWrite = 1. For lw, MemRead = 1.

R-type Instruction (add/sub/or/and):

```
Reg[IR[15-11]] = ALUOut;
```

ALUOut is stored into the register specified by IR[15-11] (MemtoReg = 0, RegWrite = 1)

Step 5: Memory Write Back (Completion)

load (lw):

```
Reg[IR[20-16]] = MDR;
```

MDR is stored into the register specified by IR[20-16] (MemtoReg = 1, RegWrite = 1, RegDst = 0)