Lab 7: Single-Cycle CPU Design

7.1 Objectives

After completing this lab, you will:

- Learn how to design a single-cycle CPU
- Verify the correct operation of your single-cycle CPU design

7.2 Subset of the MIPS Instructions included in CPU Design

In this section, we will illustrate the design of a single-cycle CPU for a subset of the MIPS instructions, shown in Table 7.1. These include the following instructions:

- ALU instructions (R-type): **add, sub, and, or, xor, slt**
- Immediate instructions (I-type): **addi, slti, andi, ori, xori**
- Load and Store (I-type): **lw, sw**
- Branch (I-type): **beq, bne**
- Jump (J-type): **j**

Although this subset does not include all the integer instructions, it is sufficient to illustrate the design of datapath and control. Concepts used to implement the MIPS subset are used to construct a broad spectrum of computers. For each instruction to be implemented, you need to identify all the steps that need to be performed for the execution of each instruction expressed in register transfer level (RTL) notation. These steps are summarized below for all the instructions to be implemented:

- R-type Fetch instruction: Instruction ← MEM[PC]

Fetch operands: data1 ← Reg(Rs), data2 ← Reg(Rt)

Execute operation: ALU_result ← func(data1, data2)

Write ALU result: Reg(Rd) ← ALU_result

Next PC address: PC ← PC + 4

Table 7.1: MIPS instructions subset implemented in CPU design.

Instruction	Meaning	Format					
add rd, rs, rt	addition	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x20
sub rd, rs, rt	subtraction	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x22
and rd, rs, rt	bitwise and	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x24
or rd, rs, rt	bitwise or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x25
xor rd, rs, rt	exclusive or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x26
slt rd, rs, rt	set on less than	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x2a
addi rt, rs, im^{16}	add immediate	0x08	rs^5	rt^5	im^{16}		
slti rt, rs, im^{16}	slt immediate	0x0a	rs^5	rt^5	im^{16}		
andi rt, rs, im^{16}	and immediate	0x0c	rs^5	rt^5	im^{16}		
ori rt, rs, im^{16}	or immediate	0x0d	rs^5	rt^5	im^{16}		
xori rt, im^{16}	xor immediate	0x0e	rs^5	rt^5	im^{16}		
lw rt, im^{16}(rs)	load word	0x23	rs^5	rt^5	im^{16}		
sw rt, im^{16}(rs)	store word	0x2b	rs^5	rt^5	im^{16}		
beq rs, rt, im^{16}	branch if equal	0x04	rs^5	rt^5	im^{16}		
bne rs, rt, im^{16}	branch not equal	0x05	rs^5	rt^5	im^{16}		
j im^{26}	jump	0x02	im^{26}				

❖ I-type Fetch instruction: Instruction ← MEM[PC]

Fetch operands: data1 ← Reg(Rs), data2 ← Extend(imm16)

Execute operation: ALU_result ← op(data1, data2)

Write ALU result: Reg(Rt) ← ALU_result

Next PC address: PC ← PC + 4

❖ BEQ Fetch instruction: Instruction ← MEM[PC]

Fetch operands: data1 ← Reg(Rs), data2 ← Reg(Rt)

Equality: zero ← subtract(data1, data2)

Branch: if (zero) PC ← PC + 4 + 4×sign_ext(imm16)

else PC ← PC + 4

❖ LW Fetch instruction: Instruction ← MEM[PC]

Fetch base register: base ← Reg(Rs)

Calculate address: address ← base + sign_extend(imm16)

Read memory: data ← MEM[address]

Write register Rt: Reg(Rt) ← data

Next PC address: PC ← PC + 4

❖ SW Fetch instruction: Instruction ← MEM[PC]

Fetch registers: base ← Reg(Rs), data ← Reg(Rt)

Calculate address: address ← base + sign_extend(imm16)

Write memory: MEM[address] ← data

Next PC address: PC ← PC + 4

- Jump Fetch instruction: Instruction ← MEM[PC]

Target PC address: target ← PC[31:28] , Imm26 , '00'

Jump: PC ← target

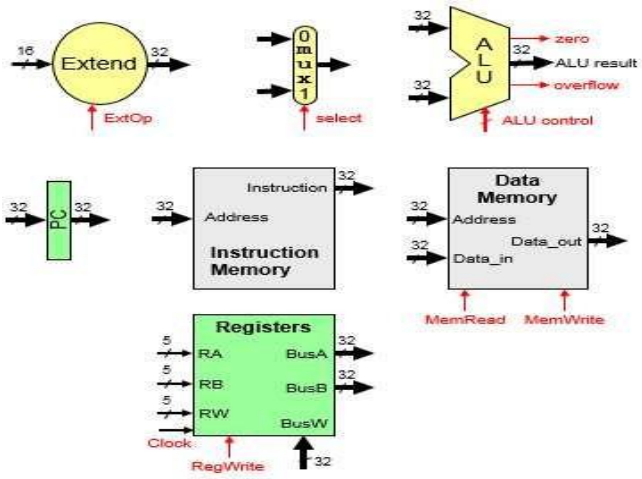
7.3 Data Path Design

The first step in designing a datapath is to determine the requirements of the instruction set in termsof components. These include the following:

- Memory
  - Instruction memory where instructions are stored
  - Data memory where data is stored
- Registers
  - 32 × 32-bit general purpose registers, R0 is always zero
  - Read source register Rs
  - Read source register Rt
  - Write destination register Rt or Rd
- Program counter PC register and Adder to increment PC
- Sign and Zero extender for immediate constant
- ALU for executing instructions

The needed components are summarized below:

- Combinational Elements
  - ALU, Adder
  - Immediate extender
  - Multiplexers
- Storage Elements
  - Instruction memory
  - Data memory
  - PC register
  - Register file



We can now assemble the datapath from its components. For instruction fetching, we need:

- ✧ Program Counter (PC) register
- ✧ Instruction Memory
- ✧ Adder for incrementing PC

The implementation of the instruction fetch process is illustrated in Figure 7.1. Since all the MIPS instructions are 32-bit instructions (i.e. each instruction is stored in 4 address locations) and since the instruction memory will be aligned on 4-byte boundary, the least significant 2-bits of instruction addresses will always be 0. Thus, it is sufficient the update the most significant 30 bits of the PC.

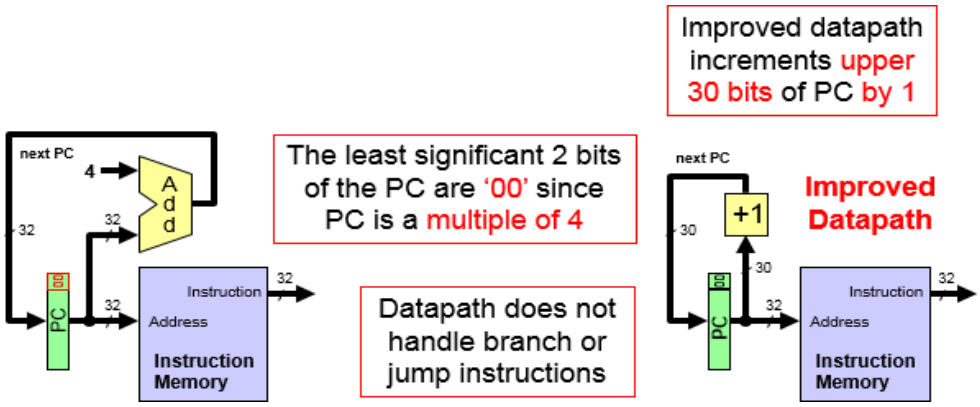


Figure 7.1: Data path component for instruction fetching.

To execute R-type instructions, we need to read the content of registers Rs and Rt, perform an ALU operation on their contents and then store the result in the register file to register Rd. The datapath for executing R-type instructions is shown in Figure 7.2.

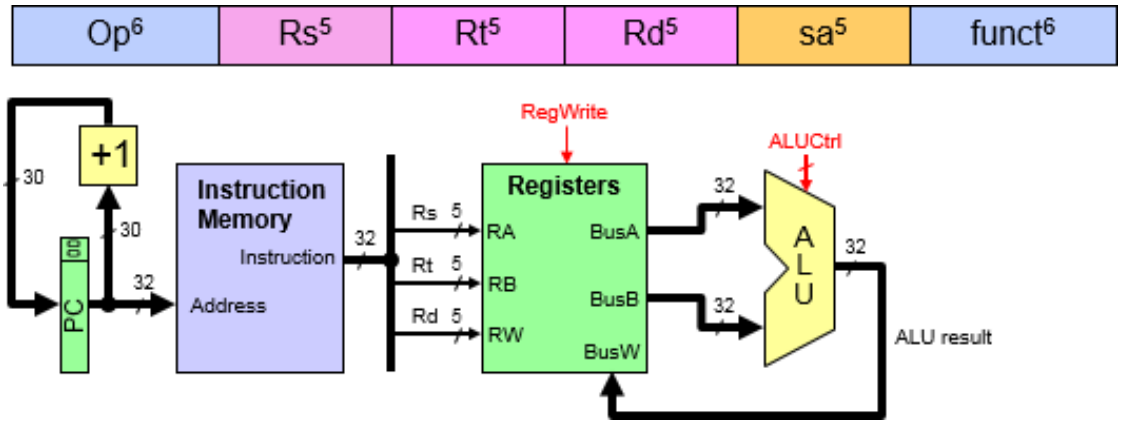


Figure 7.2: Data path implementation of R-type instructions.

The control signals needed for the execution of R-type instructions are:

- ALUCtrl is derived from the funct field because Op = 0 for R-type
- RegWrite is used to enable the writing of the ALU result

The execution of the I-type instructions is similar to the R-type instructions with the difference that the second operand is an immediate value instead of a register and that the destination register is determined by Rt instead of Rd. The 16-bit immediate value needs to be extended to a 32-bit value by either adding 16 0's or by extending the sign bit. The datapath for the execution of I-type instructions is given in Figure 7.3.

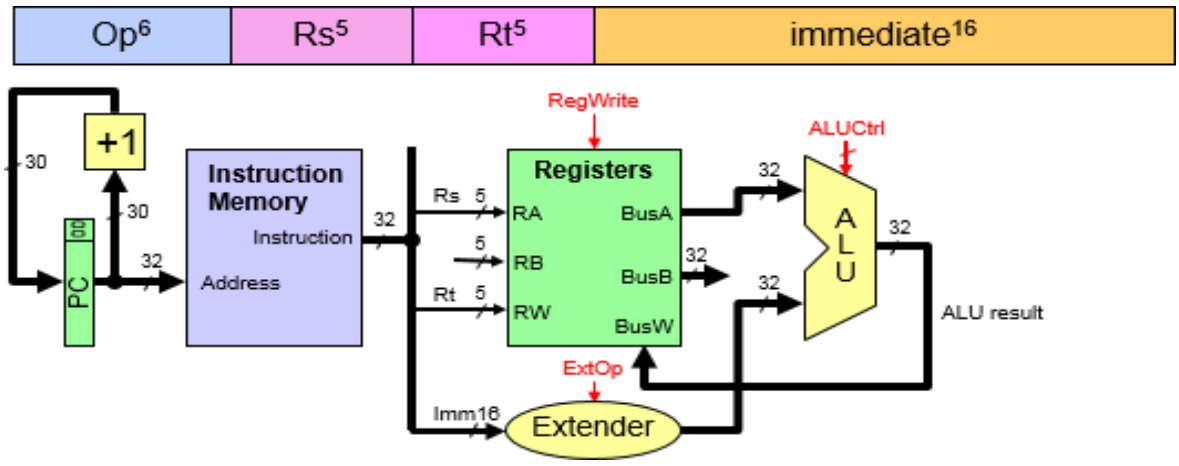


Figure 7.3: Data path implementation of I-type instructions.

The control signals needed for the execution of I-type instructions are:

- ALUCtrl is derived from the Op field
- RegWrite is used to enable the writing of the ALU result
- ExtOp is used to control the extension of the 16-bit immediate

Next we combine the datapath for executing both the R-type and I-type instructions as shown in Figure 7.4. A multiplexer is added to select between Rd and Rt to be connected to Rw in the register file to determine the destination register. Another multiplexer is added to select the second ALU input as either the source register Rt data on BusB or the extended immediate.

The control signals needed for the execution of either R-type or I-type instructions are:

- ALUCtrl is derived from either the Op or the funct field
- RegWrite enables the writing of the ALU result
- ExtOp controls the extension of the 16-bit immediate
- RegDst selects the register destination as either Rt or Rd
- ALUSrc selects the 2nd ALU source as BusB or extended immediate

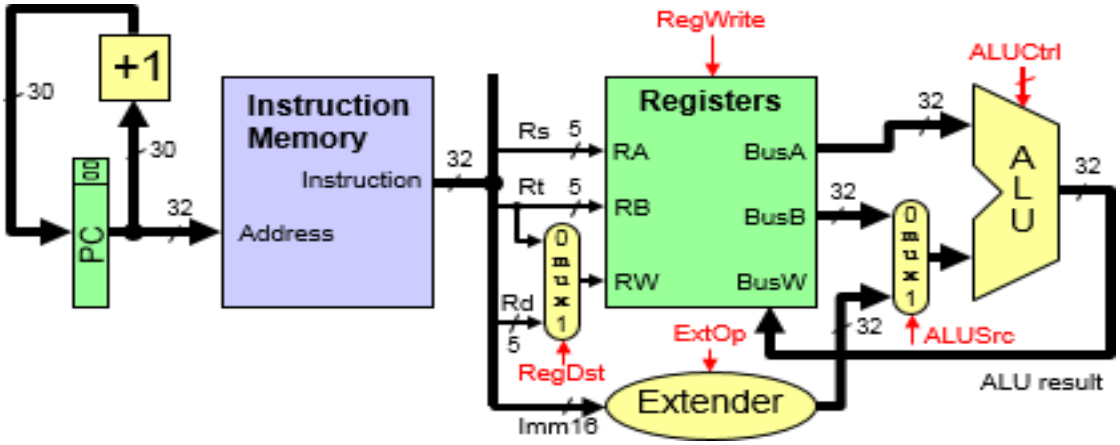


Figure 7.4: Data path implementation of R-type and I-type instructions.

To execute the load and store instructions, we need to add data memory to the datapath. For the load and store instructions, the ALU will be used to compute the memory address by adding the content of register Rs coming through BusA and the sign-extended immediate value. For the load instruction, we need to write the output of the data memory to register file. Thus, a third multiplexeris added to select between the output of the ALU and the data memory to be written to the register file. BusB is connected to Datain of Data Memory for store instructions. The updated CPU with the capability for executing load and store instructions is shown in Figure 7.5.

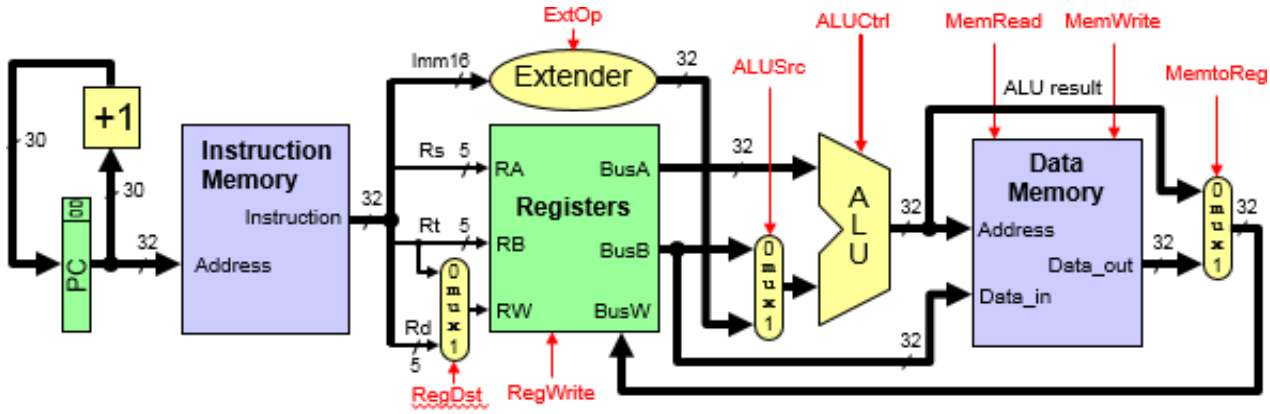


Figure 7.5: Data path implementation with load/store instructions.

The additional control signals needed for the execution of load and store instructions are:

- ✧ MemRead for load instructions
- ✧ MemWrite for store instructions
- ✧ MemtoReg selects data on BusW as ALU result or Memory Data_out

For executing jump and branch instructions, we need to add a block, called NextPC, to compute the target address. In addition, we need to add a multiplexer to select the input to the PC register to be either the incremented PC address or the target address generated by NextPC block. For branch

instructions, the ALU is used to perform subtract operation to subtract the content of the two compared registers Rs and Rt. The updated data path to include the execution of the jump and branch instructions is given in Fig 7.6.

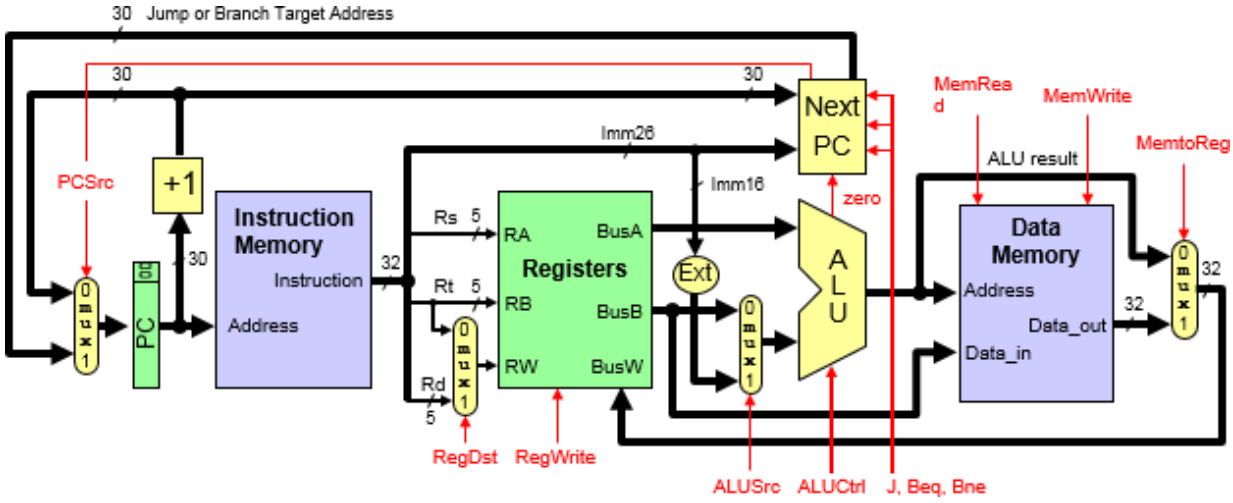


Figure 7.6: Data path implementation with jump/branch instructions.

The additional control signals needed for the execution of jump and branch instructions are:

- J, Beq, Bne for jump and branch instructions
- Zero condition of the ALU is examined
- PCSrc = 1 for Jump & taken Branch

The details of the NextPC block are illustrated in Fug. 7.7. For the jump instruction, the target address is computed by concatenating the upper 4 bits of PC with Imm26 (i.e. the 26-bit immediate value). However, for branch instructions the target address is computed by adding the sign-extended version of the 16-bit immediate value with the incremented value of PC. Note that the immediate value is computed by the assembler as [Terget – (PC + 4)]/4. Thus, to restore the target address we need to multiply the immediate value by 4 (i.e. shift it 2 bits to the left) and then add PC+4 to it. Since we are updating the most significant 30-bits of PC, this is achieved by adding PC+1 to the immediate value. The PCSrc signal is set when a branch instruction is taken or a jump instruction is executed, which is implemented by the equation PCSrc = J + (Beq . Zero) + (Bne . Zero').

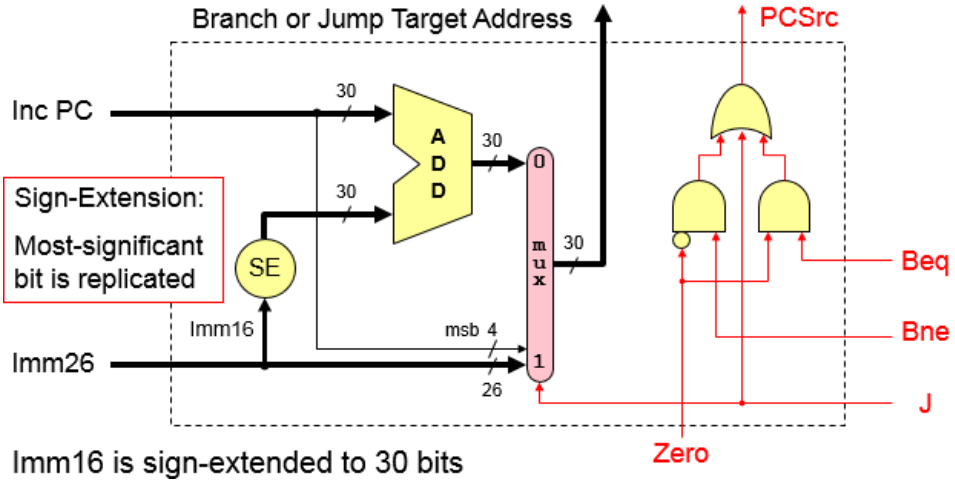


Figure 7.7: Implementation of Next PC block.

7.4 Control Unit Design

The control unit of the single-cycle CPU can be decomposed into two parts Main Control and ALU Control. The Main Control unit receives a 6-input opcode and generates all the needed control signals other than the ALU control. However, the ALU Control gets a 6-bit function field from the instruction and ALUCtrl signal from the Main Control. The single cycle CPU including the datapath and control unit is illustrated in Figure 7.8.

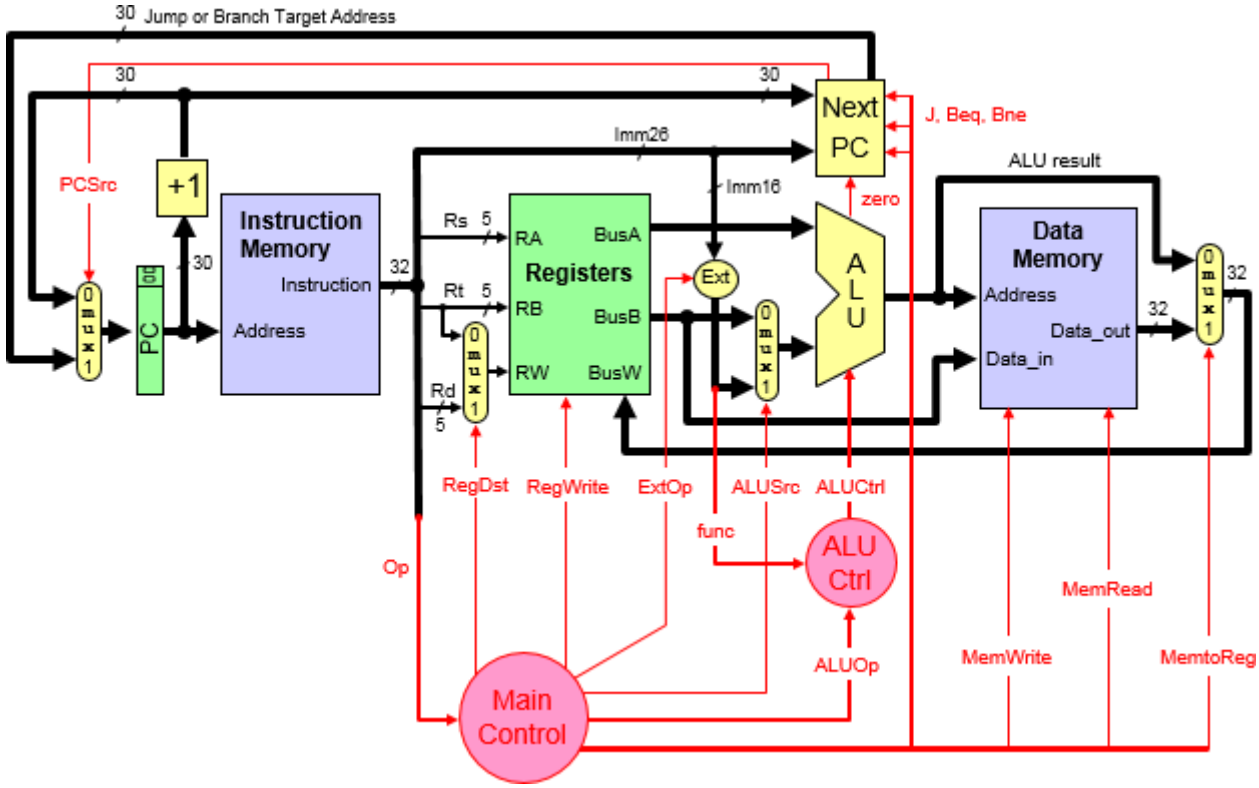


Figure 7.8: Single-cycle CPU.

To design the Main Control unit, we need to generate the control table which lists for each instruction, the control values needed to execute the instruction. This is illustrated in Table 7.2.

Table 7.2: Main Control Signal Values.

Op	Reg Dst	Reg Write	Ext Op	ALU Src	ALU Op	Beq	Bne	J	Mem Read	Mem Write	Mem toReg
R-type	1 = Rd	1	x	0=BusB	R-type	0	0	0	0	0	0
addi	0 = Rt	1	1=sign	1=Imm	ADD	0	0	0	0	0	0
slti	0 = Rt	1	1=sign	1=Imm	SLT	0	0	0	0	0	0
andi	0 = Rt	1	0=zero	1=Imm	AND	0	0	0	0	0	0
ori	0 = Rt	1	0=zero	1=Imm	OR	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
sw	x	0	1=sign	1=Imm	ADD	0	0	0	0	1	x
beq	x	0	x	0=BusB	SUB	1	0	0	0	0	x
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

One we have the Control Table, the control unit can be design easily using a 6x64 decoder that has the 6-bit opcode as input and a signal for each instruction as output. Then each control signal willbe either an OR gate of the instructions signals that make this signal 1 or a NOR gate of the instructions signals that make this signal 0, which ever results in a smaller gate size. The decoder and the logic equations for the Main Control signals are shown in Figure 7.9.

$$\text{RegDst} <= \text{R-type}$$

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

$$\text{ExtOp} <= \overline{(\text{andi} + \text{ori} + \text{xori})}$$

$$\text{ALUSrc} <= \overline{(\text{R-type} + \text{beq} + \text{bne})}$$

$$\text{MemRead} <= \text{lw}$$

$$\text{MemWrite} <= \text{sw}$$

$$\text{MemtoReg} <= \text{lw}$$

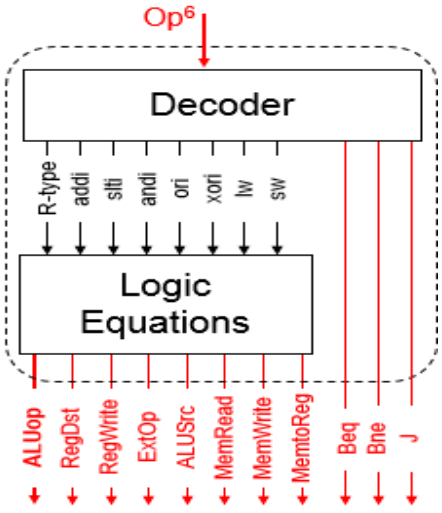


Figure 7.9: Main control unit design.

Similarly, the ALU control signals equations can be derived based on the 6-bit function field and the ALUOp signal generated by the Main Control unit.

It should be observed that the control unit signals equation can also be derived using K-map technique without using a decoder. However, using a decoder makes the design of the control unit simple.

7.5 In-Lab Tasks

1. For the instructions in the CPU that you are going to design, list all the steps that are neededfor the execution of each instruction in RTL notation.
2. Ensure that you have all the needed components for constructing your datapath.
3. Design the datapath for your CPU and model it using logisim.
4. Apply the needed values for the control signals needed for the execution of each instructionto ensure correct functionality of the datapath.
5. Design the control unit of your CPU and model it using logisim.
6. Test the correct functionality of the control unit by ensuring that it generates the correctcontrol signal values for each instruction.
7. Model the single cycle CPU design in logisim by combining the datapath and control units.
8. Test the correct functionality of your CPU by storing all the implemented instructions in theinstruction memory and verifying the correct execution of each instruction.