**CDA 4205 Project Part II: Single-Cycle CPU Design**

1. **Objectives**

Using the Logisim Simulator to design a single-cycle CPU for a subset of the MIPS instructions and verify the correct operation of the designed single-cycle CPU.

1. **Tasks**
2. Design the datapath for a single-cycle CPU and model it using logisim.
3. Apply the needed values of for the control signals needed for the execution of each instruction to ensure correct functionality of the datapath.
4. Design the control unit for the designed datapath and model it using logisim.
5. Test the correct functionality of the control unit by ensuring that it generates the correct control signal values for each instruction.
6. Model the single cycle CPU design in logisim by combining the datapath and control units.
7. Test the correct functionality of the designed CPU by storing all the implemented instructions in the instruction memory and verifying the correct execution of each instruction.
8. **Subset of the MIPS Instructions Required for the Designed CPU**

The designed single-cycle CPU only implements a subset of the MIPS instructions as shown in Table I. This subset instructions are sufficient to illustrate the design of datapath and control unit and the required subset includes 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 instruction (I-type): lw, sw
* Branch (I-type): beq, bne
* Jump (J-type): j

1. **Instruction RTL Notation**

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

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

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

Execute operation: ALU\_result ← op(data1, data2)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Meaning** | **Format** | | | | | |
| add rd, rs, rt | addition | op6 = 0 | rs5 | rt5 | rd5 | 0 | 0x20 |
| sub rd, rs, rt | subtraction | op6 = 0 | rs5 | rt5 | rd5 | 0 | 0x22 |
| and rd, rs, rt | bitwise and | op6 = 0 | rs5 | rt5 | rd5 | 0 | 0x24 |
| or rd, rs, rt | bitwise or | op6 = 0 | rs5 | rt5 | rd5 | 0 | 0x25 |
| xor rd, rs, rt | exclusive or | op6 = 0 | rs5 | rt5 | rd5 | 0 | 0x26 |
| slt rd, rs, rt | set on less than | op6 = 0 | rs5 | rt5 | rd5 | 0 | 0x2a |
| addi rt, rs, imm16 | add immediate | 0x08 | rs5 | rt5 | imm16 | | |
| slti rt, rs, imm16 | slt immediate | 0x0a | rs5 | rt5 | imm16 | | |
| andi rt, rs, imm16 | and immediate | 0x0c | rs5 | rt5 | imm16 | | |
| ori rt, rs, imm16 | or immediate | 0x0d | rs5 | rt5 | imm16 | | |
| xori rt, imm16 | xor immediate | 0x0e | rs5 | rt5 | imm16 | | |
| lw rt, imm16(rs) | load word | 0x23 | rs5 | rt5 | imm16 | | |
| sw rt, imm16(rs) | store word | 0x2b | rs5 | rt5 | imm16 | | |
| beq rs, rt, offset16 | branch if equal | 0x04 | rs5 | rt5 | imm16 | | |
| bne rs, rt, offset16 | branch if not equal | 0x05 | rs5 | rt5 | imm16 | | |
| j imm26 | jump | 0x02 | Imm26 | | | | |

**Table I: MIPS instructions subset implemented in CPU design.**

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

1. **Datapath Components**

The first step in designing a datapath is to determine the requirements of the instruction set in terms of 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, $0 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
* Multiplexers
* To select the write destination register Rt or Rd
* To select the ALU operand inputs from register file or immediate operand
* To update PC address with PC + 4 or branch destination or jump destination.

1. **Assemble Datapath From its Components**
2. Implementation of the Instruction Fetching

For instruction fetching, we need Program Count (PC) register, instruction memory and 32-bit full-adder for incrementing PC. PC is used as the instruction memory address to read instruction from the instruction memory, then PC is incremented by 4 to point to the next instruction to be executed. Since the instruction memory is aligned on 4-byte boundary, the least significant 2-bits of instruction addresses will always be 0. Thus, we can use 30-bit full-adder to update the most significant 30 bits of the PC.

1. R-type Instruction Execution

To execute R-type instructions, we need to read the content of registers Rs and Rt. The register numbers for Rs and Rt are specified in the instruction. Then, need to perform an ALU operation, on the contents of Rs and Rt. The ALU operation is specified by the function field in the instruction. Finally, we need to store the result in the register file to register Rd.

The control signals needed for the execution of R-type instructions include ALUCtrl and RegWrite. ALUCtrl is derived from the function field and RegWrite is used to enable the writing of the ALU result in the register file.

1. I-type Instruction Execution

The I-type instruction execution is quite similar to the R-type instructions. The differences between R-type instruction execution and I-type instruction execution include:

* In R-type instruction, the second operand comes register Rt while in I-type instruction the second operand comes from the 16-bit immediate constant value in the instruction.
* The destination register is specified by Rd and Rt in R-type instruction and I-type instruction, respectively.

Based on the implementation of R-type instruction execution, the following components are needed to execution both R-type and I-type instructions:

* 16-bit to 32 bit extender: ALU data operand inputs need to be 32 bits, hence, 16-bit immediate value needs to be either zero extended or sign extended to a 32-bit value dependent on the opcode in the I-type instruction. To selection between perform zero-extension and sign-extension, 1-bit control signal ExtOp is need to control the extension of the 16-bit immediate value to 32-bit value, which will be forwarded to ALU second data operand input.
* Destination Register Selection Multiplexer: in order to determine the destination register, a multiplexer is needed to select either Rd or Rt connecting to Rw in the register file. A 1-bit control signal RegDst is needed to select the destination register between Rt and Rd. Also, another 1-bit control signal RegWrite is needed to enable the writing operation of the ALU result into the destination register.
* ALU Data Source Selection Multiplexer: a multiplexer is needed to select the second ALU input as either the source register Rt data on BusB or the extended immediate. Another 1-bit control signal ALUSrc is needed to select the 2nd ALU data operand input between Rt data on BusB or extended immediate value from the output of the 16-bit to 32-bit extender.

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

* ALUCtrl is derived from either the Opcode for I-type instructions or the opcode and funct field for R-type instructions.
* RegWrite is used to enable the writing operations of the ALU result.
* ExtOp is used to control the zero-extension or sign-extension of 16-bit immediate value to 32-bit value.
* RegDst selects the register destination as either Rt or Rd.
* ALUSrc selcts the second ALU data operand input as either Rt data on BusB or the extended immediate.

1. Load/store Instruction Execution

In order to execute the load and store instructions, we need to add data memory to the datapath. For the load and store instruction, the ALU is used to compute the memory address by adding the content of base register, Rs, and the sign-extended immediate value. The ALU result, which is the memory address, is connected to data memory address bus.

* Load instruction: For the load instruction, the output of the data memory will be written to register file through BusW. For R-type or I-type instruction, ALU result will be written to register file through BusW also. Therefore, a third multiplexer is added to select writing content between output of the ALU and the data memory. A 1-bit control signal is needed to select the data on BusW as either ALU result or memory Data\_out.
* Store instruction: For the store instruction, the register content on BusB from the register file will be written into the data memory. Hence, BusB is connected to Data\_in of data memory.

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

* MemRead enables read operation of data memory for load instructions.
* MemWrite enables write operation of data memory for store instructions.
* MemtoReg selects data on BusW as ALU result or Data Memory Data\_out.

1. J-type Instruction Execution

To execute jump and branch instructions, we need to compute the target address.

* Jump instruction: the target address is computed by concatenating the upper 4 bits of PC with the 26-bit immediate value from the instruction, and appending 2 bits 0s.
* Branch instructions, the target address is calculated by adding the sign-extended 16-bit immediate value with the incremented value of PC, PC + 4 + 16-bit immediate \* 4. Therefore, we need to shift the 16-bit immediate value 2 bits to the left and then add PC + 4 to it. We can only update the most significant 30-bits of PC by adding PC + 1 to the immediate value and the least significant 2-bits of PC is always set to 0. For branch instructions, ALU is used to calculate the difference between the content of the two comparing registers Rs and Rt.

To select input to the PC register to be either the incremented PC address or the calculated target address from jump or branch instructions. Two additional multiplexers A and B are needed. Multiplexer A is used to select the target address from jump or branch instruction. Multiplexer B is used to select the input to the PC register to be either the incremented PC address or the calculated target address from jump or branch instructions.

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

* J is the control signal for jump instruction.
* Beq and Bne are the control signals for branch instructions.
* Zero represents the ALU result is 0. This control signal represents the comparing result of two comparing registers Rs and Rt.
* PCSrc is used to select the input to the PC register to be either the incremented PC address or the calculated target address from jump or branch instructions. PCSrc = 1 represents the instruction is Jump instruction or taken branch instructions. PCSrc = 0 represents the PC will be updated with the incremented PC address.

More details on two multiplexers:

* Multiplexer A: two inputs to multiplexer A are the calculated jump target address and the calculated branch target address, and the control signal J is used to select the calculated target address. J = 1, the jump target address is selected; otherwise, the branch target address is selected.
* Multiplexer B: two inputs to multiplexer B are the incremented PC address or the output of multiplexer A and the control signal PCSrc is used as the selection signal. PCSrc is set when a branch instruction is taken or a jump instruction is executed, which is implemented by the Boolean function.

1. **Control Unit Design**

The control unit of the single-cycle CPU can be divided into two parts, Main control and ALU control.

1. **Main Control**

The main control unit receives a 6-bit opcode and generates all the needed control signals other than the ALU control signals. To design the Main Control unit, we need to generate the control table which lists the required control values for executing each instruction.

To implement the control unit, we need a 6 x 64 decoder that has the 6-bits opcode as the input and the control signals in the above table as outputs. The Boolean functions for the control signals are listed as follows: (Note: in the following equations, the right hand represents the decoder output)

RegDst = R-type

RegWrite = (sw + beq + bne + j)’

ExtOp = (andi + ori + xori)’

ALUSrc = (R-type + beq + bne)’

MemRead = lw

MemWrite = sw

MemtoReg = lw

The control signals of Beq, Bne and J can be forwarded directly from the decoder outputs.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Op** | **R-type** | **addi** | **slti** | **andi** | **ori** | **xori** | **lw** | **sw** | **beq** | **bne** | **j** |
| **RegDst**  **(1 = Rd)**  **(0 = Rt)** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | X |
| **RegWrite** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| **ExtOp**  **(1 = sign)**  **(0 = zero)** | X | 1 | 1 | 0 | 0 | 0 | 1 | 1 | X | X | X |
| **ALUSrc**  **(0 = BusB)**  **(1 = Imm)** | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | X |
| **ALUOp** | R-type | ADD | SLT | AND | OR | XOR | ADD | ADD | SUB | SUB | X |
| **Beq** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| **Bne** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **J** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **MemRead** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| **MemWrite** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| **MemtoReg** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | X | X | X | X |

1. **ALU Control**

The ALU Control unit receives a 6-bit function field from the instruction and ALUOp signal from the Main Control. ALUOp indicates whether the operation to be performed should be ADD (100), SUB (101), AND (001), OR (010), XOR (011), SLT (110) or determined by the operation encoded in the funct field (000). Similar to the procedure to generate the control signal Boolean functions for the main control unit, 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.

|  |  |  |  |
| --- | --- | --- | --- |
| **Op** | **Funct** | **ALU Operation** | **4-bit ALU Control** |
| R-type | AND | AND | 0001 |
| R-type | OR | OR | 0010 |
| R-type | XOR | XOR | 0011 |
| R-type | ADD | ADD | 0100 |
| R-type | SUB | SUB | 0101 |
| R-type | SLT | SLT | 0110 |
| ADDI | X | ADD | 0100 |
| SLTI | X | SLT | 0110 |
| ANDI | X | AND | 0001 |
| ORI | X | OR | 0010 |
| XORI | X | XOR | 0011 |
| LW | X | ADD | 0100 |
| SW | X | ADD | 0100 |
| BEQ | X | SUB | 0101 |
| BNE | X | SUB | 0101 |
| J | X | X | X |