MICROPROCESSOR ARCHITECTURE AND IMPLEMENTATION

Introduction

We have examined the SystemVerilog descriptions of combinational and sequential building blocks used in digital systems, including arithmetic circuits, counters, register files, and memory arrays. The ELEC0010 lab will involve combining these building blocks to form a microprocessor microarchitecture. This document provides a recap of the lecture course in the first-year course ELEC0004 Digital Electronics 1, and is a reference guide for your lab activities.

The first step in designing a microprocessor is to define the architecture of the computer. The architecture is the programmer's view of a computer. It is defined by the instruction set (the language) and operand locations (registers and memory).

The words in a computer's language are called instructions. The computer's vocabulary is called the instruction set. All programmes running on a computer use the same instruction set. Even complex software applications are compiled into a series of simple instructions such as add, subtract, and jump. Computer instructions indicate both the operation to perform and the operands (the values) to use in the operation. The operands may come from registers or from the instruction itself.

A computer architecture doesn't define the underlying hardware implementation. Often, many different hardware implementations of a single architecture exist. For example, Intel and AMD sell various microprocessors based on the same x86 architecture. They can all run the same programmes, but have different underlying hardware. The underlying hardware is called the microarchitecture.

In this section, we will start by examining a simple computer architecture, and then examine a microarchitecture implementation of it.

1 INSTRUCTION SET ARCHITECTURE

Assembly language is the human-readable representation of the computer's native language, which is called machine language. Each assembly language instruction specifies both the operation to perform and the operands which will be operated on. We'll start by looking at simple arithmetic instructions and show how these operations are written in assembly language. We'll then define the instruction operands which are stored in registers or as constants in the instruction itself.

1.1 Assembly language instructions

The codes below are for arithmetic operations on b and c with the results written into a. The code shown on the left is in a high-level language (using the syntax of e.g. C, C++, or Java), and the equivalent assembly language code is written on the right.

High-Level Code	Assembly Code
a = b + c; a = b – c;	add a, b, c sub a, b, c

The first part of the assembly instruction, e.g. *add*, is called the mnemonic and indicates what operation is to be performed. The operation is performed on b and c (the source operands), and the result is written into a (the destination operand). Instructions operate on, at most, two source operands. More complex high-level code translates into multiple assembly language instructions, as shown in the example below:

High-Level Code	Assembly Code
a = b + c - d; // Complex high-level code	sub t, c, d # t = c – d add a, b, t # a = b + t

Note that, in the high-level language example, comments begin with //, while in assembly language, they begin with #.

1.2 Operands: Registers and Constants

In the assembly code examples above, the variables a, b, c and t are all operands. But computers operate on 1's and 0's, not variable names. The instructions need physical locations from which to retrieve the binary data, and to which the resulting binary data can be stored. Operands can be variables stored in binary form in registers, or they may be constants stored in the instruction itself.

We will consider an 8-bit computer architecture. This means it uses 8-bits to represent the operands.

Registers

Most computer architectures specify a number of registers that hold the operands. In the architecture we are considering, there are 16 registers, forming what is called the register file. Each register can store an 8-bit word. The register file can be implemented as a 16 × 8-bit static random-access memory (SRAM) array.

In assembly code, the register names are preceded by the letter x. The 16 registers in the register file are referred to as x0, x1, x2, x3, … x15. Registers x1 to x15 inclusive can be written to by the user. Register x0 always holds the value 0 and cannot be written to, as 0 is a useful value to have readily available. The example below shows the *add* instruction with the variables a, b and c placed in registers x1, x2 and x3.

High-Level Code	Assembly Code
a = b + c;	# x1 = a, x2 = b, x3 = c add x1, x2, x3

Constants/immediates

In assembly language, constants contained within the instruction are called immediates (since they are immediately available from the instruction itself, and don't require a register access). *Add immediate* (*addi*) is a common assembly language instruction that uses an immediate operand.

Examples of code using immediate instructions are shown below.

High-Level Code	Assembly Code
a = b + 4;	# x1 = a,x 2 = b addi x1, x2, 4
a = b – 12;	# x3 = a, x4 = b addi x3, x4, -12
a = 9;	# x7 = a addi x7, x0, 9

1.3 Machine language

Assembly language is convenient for humans to read. However, digital circuits understand only 1's and 0's. Therefore, a programme written in assembly language is translated from mnemonics and register names to a representation using only 1's and 0's, called machine language.

The architecture we are considering uses 24-bit instructions ($Instr_{23:0}$), consisting of:

- a 4-bit field indicating the operation (this is called the op code),
- three 4-bit fields containing the three register addresses (the destination register and the two source registers)
- an 8-bit field containing an immediate value

The instruction format is shown below.

operation code	destination register	source register 1	source register 2	immediate value
4 bits	4 bits	4 bits	4 bits	8 bits
[23:20]	[19:16]	[15:12]	[11:8]	[7:0]

This instruction format allows up to $2^4 = 16$ possible operations, and allows all $2^4 = 16$ registers in the register file to be addressed. The op code for the *add* operation is 0010, for the *sub* operation it's 0011, and for the *addi* operation, 0110 is used. The full set of operations and the corresponding op codes for this architecture will be listed later on.

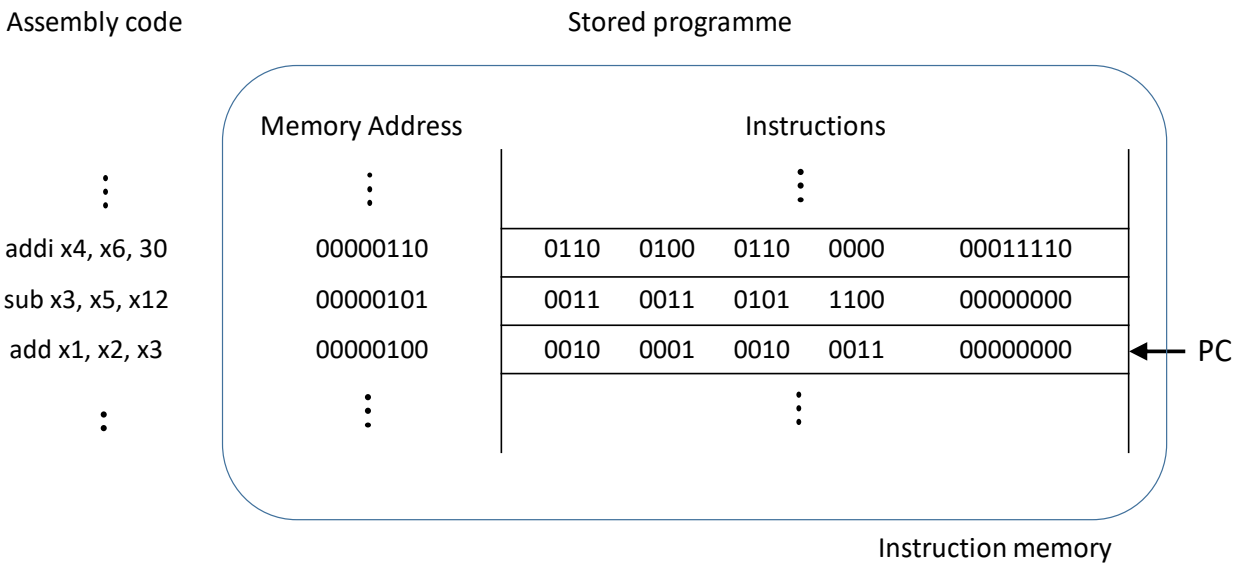
Some examples of assembly code translated into machine code are given in the table below:

Assembly code	**Machine code**				
	op code	destination register	source register 1	source register 2	immediate
addi x4, x6, 30	0110	0100	0110	0000	00011110
sub x3, x5, x12	0011	0011	0101	1100	00000000
add x1, x2, x3	0010	0001	0010	0011	00000000

Unused fields (e.g., the immediate field in the *add* and *sub* instructions, and the source register 2 field in the *addi* instruction) can be simply set to zero. The machine code can also be represented in hex (with six digits per instruction), as follows.

Assembly code	**Machine code**				
	op code	destination register	source register 1	source register 2	immediate
addi x4, x6, 29	6	4	6	0	1 E
sub x3, x5, x12	3	3	5	C	0 0
add x1, x2, x3	2	1	2	3	0 0

The instructions are stored in a programmable memory array, called the instruction memory. The instruction memory has an 8-bit address, and its size is 256 × 24 bits. The figure below shows how machine instructions are stored in memory:



Program memory

To run or execute the programme, the processor fetches the instructions from memory sequentially (one after the other). Each fetched instruction is decoded and executed by the digital hardware. The address of the current instruction is kept in an 8-bit register called the programme counter (PC). To execute the lines of code in the example above, the value of *PC* is set to the address of the first line of the code, 00000100 (which can be conveniently written in hexadecimal as 0x04). The processor reads (*fetches*) the instruction at that memory address and executes the instruction (*add* x1, x2, x3). The processor then increments the value of the programme counter (PC) to the next memory address (0x05), fetches that instruction (*sub* x3, x5, x12) and executes it, and keeps repeating this process.

The architecture we are considering has 8 instructions, performing arithmetic operations (addition and subtraction on two variables, addition on one variable and one constant), logical operations (bit-wise AND and OR, either on two variables, or on one variable and one constant), and a conditional branch operation (the programme counter jumps to a new value in the case of two variables being equal). The detailed description of each of these instructions, and the corresponding assembly code mnemonics and op codes, are listed in the table below (In this table, rs1 and rs2 refer to source register 1 and source register 2, respectively, and rd refers to the destination register).

instruction	mnemonic	op code	description
AND	and	0000	rd = rs1 AND rs2; pc = pc + 1
OR	or	0001	rd = rs1 OR rs2; pc = pc + 1
add	add	0010	rd = rs1 + rs2; pc = pc + 1
subtract	sub	0011	rd = rs1 - rs2; pc = pc + 1
AND immediate	andi	0100	rd = rs1 AND immediate; pc = pc + 1
OR immediate	ori	0101	rd = rs1 OR immediate; pc = pc + 1
add immediate	addi	0110	rd = rs1 + immediate; pc = pc + 1
branch if equal	beq	0111	if rs1 == rs2 pc = immediate; else pc = pc + 1

1.4 Assembly language programming

Conditional branching

The example below shows the operation of the conditional branching, *beq*, instruction in an assembly language programme:

Instruction memory address	Assembly program	
0x00	addi x1, x0, 6	# Set the value in register x1 = 0 + 6 = 6
0x01	addi x2, x0, 12	# x2 = 0 + 12 = 12
0x02	sub x3, x2, x1	# x3 = x2 – x1 = 12 – 6 = 6
0x03	beq x1, x3, 5	# x1 == x3 == 6, so branch is taken
0x04	addi x4, x2, –9	# not executed
0x05	add x4, x2, x1	# x4 = x2 + x1 = 12 + 6 = 18
0x06	beq x0, x0, 5	# unconditional branch

When the programme in the example above reaches the branch-if-equal instruction (*beq*), the value stored in register x1 equals the value in register x3, and hence the branch is taken. That is, the next instruction executed is the *add* instruction at address 0x05 (as specified by the immediate value in the *beq* instruction). The instruction directly after the branch is not executed.

TOPIC 2 COMPUTER MICROARCHITECTURE

In this section, the design of the computer's hardware implementation (its microarchitecture) will be described. It is relatively straightforward, and we have covered all the required digital building blocks. You are familiar with circuits for arithmetic and memory, and you have learnt about computer architecture, which specifies the programmer's view of the computer.

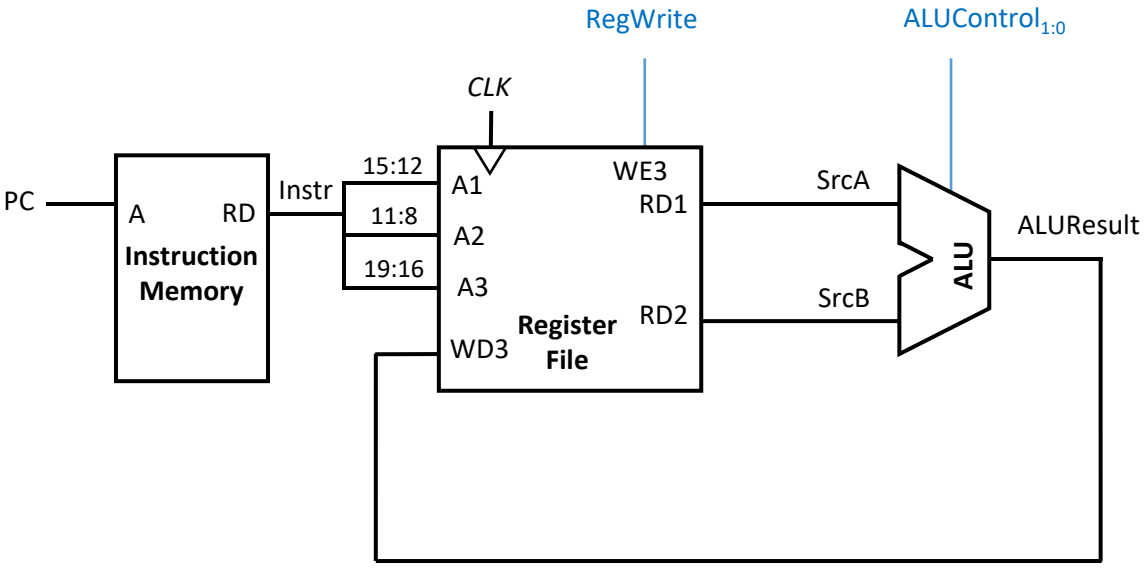
A computer's microarchitecture is the specific arrangement of registers, arithmetic and logic units (ALUs), finite state machines, memories and other logic building blocks needed to implement an architecture. A particular architecture may have many different microarchitectures, each with different trade-offs of performance, cost and complexity. We will consider an implementation of the architecture described in Topic 1 which carries out one instruction per clock cycle.

We divide our microarchitecture into two interacting parts: the datapath and the control unit. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. Our processor has an 8-bit architecture, so we will use an 8-bit datapath (i.e., the registers' and ALU's width is 8 bits). The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction, using the following control signals:

- ALU control signals
- multiplexer select signals
- register write enable signals

2.1 Datapath circuit design to implement *add*, *sub*, *and*, *or* instructions

As the starting point, consider the datapath circuit design shown below, which can perform arithmetic and logic operations on two variables, stored in the register file, and write the result into the register file:



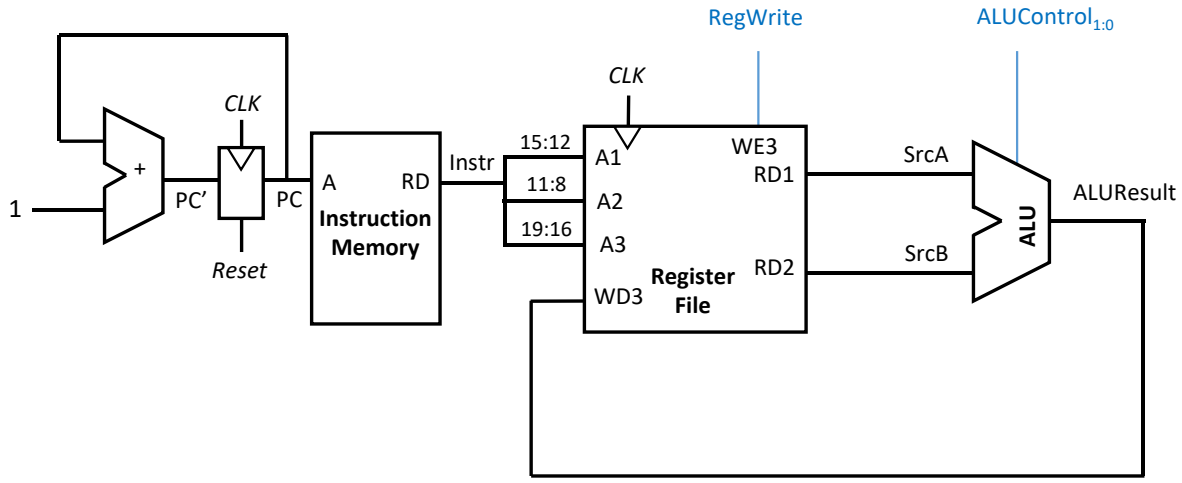
Datapath circuit design for arithmetic and logic operations on two variables

The instruction memory is on the left. Its address input, *A*, has the value *PC* (programme counter) applied to it, so the instruction ($Instr_{23:0}$) at memory address *PC* is being output. The fields in the instruction, $Instr_{15:12}$ and $Instr_{11:8}$, which specify the addresses of the two source registers, are connected to the register file read address inputs *A1* and *A2*. The contents of the registers with addresses *A1* and *A2* are output and applied to the inputs of the ALU, *SrcA* and *SrcB*, respectively (*Src* is short for Source). The function performed by the ALU is controlled by the 2-bit control signal $ALUControl_{1:0}$. The functions of the ALU are listed below.

$ALUControl_{1:0}$	Operation	Description
00	a & b	Bitwise logical operation: a AND b
01	a ^ b	Bitwise logical operation: a OR b
10	a + b	Arithmetic operation: addition: a + b
11	a – b	Arithmetic operation: subtraction: a – b

If the control signal *RegWrite*, applied to the register file write enable input (WE3), is asserted, then, at the next positive clock edge, the result from the ALU (*ALUResult*) is written into the register at address *A3*. The value of address *A3* is specified in the instruction's destination register address field ($Instr_{19:16}$). If *RegWrite* == 0, nothing is written into the register file.

In addition to the result from the ALU being written into the register file at the positive clock edge, we want the value of *PC* to be increased by 1 during this same clock edge, so that the next instruction will be fetched from memory and executed. This can be done using a counter circuit. The figure below shows the same hardware as above, but with the counter included to increment the value of *PC*. The PC register is updated with the new value, *PC*' = *PC* + 1, on each positive clock edge:

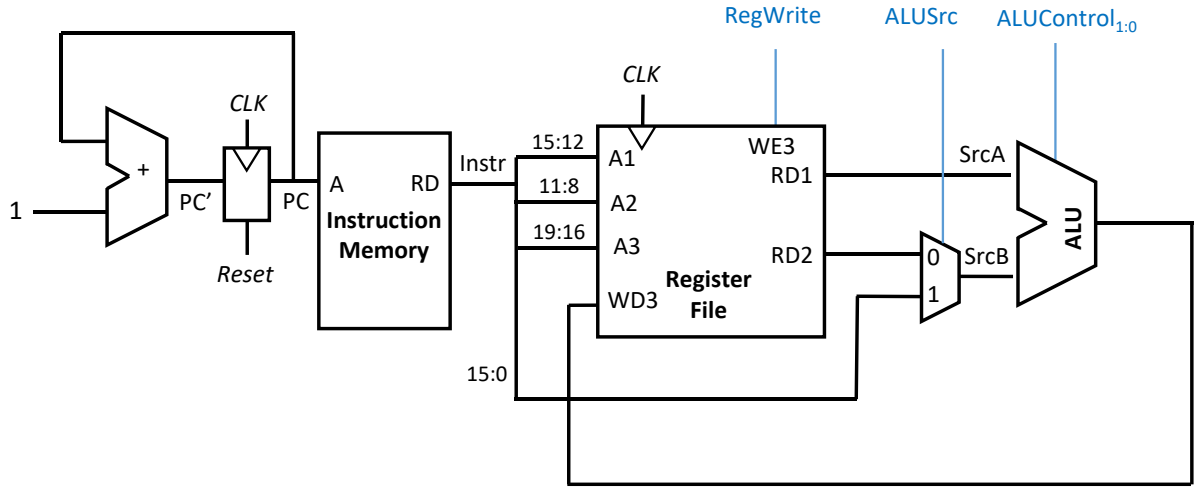


Datapath circuit design for arithmetic and logic operations on two variables, with a counter to increment value of PC

We now have a datapath circuit design which sequentially carries out instructions, stepping sequentially from one instruction to the next in the memory on each positive clock edge. The datapath is able to perform *add*, *sub*, *and*, *or* operations on two variables stored in the register file.

2.2 Extending the datapath circuit design to also implement *addi*, *andi*, *ori* instructions

We will now extend the above design so that it can carry out the immediate arithmetic and logic operations *addi*, *andi* and *ori* (which operate on one variable stored in the register file together with the immediate value contained within the instruction $Instr_{7:0}$). This extended design is shown below:



Datapath circuit design for arithmetic and logic operations on two variables, and for arithmetic and logic operations on a variable and a constant

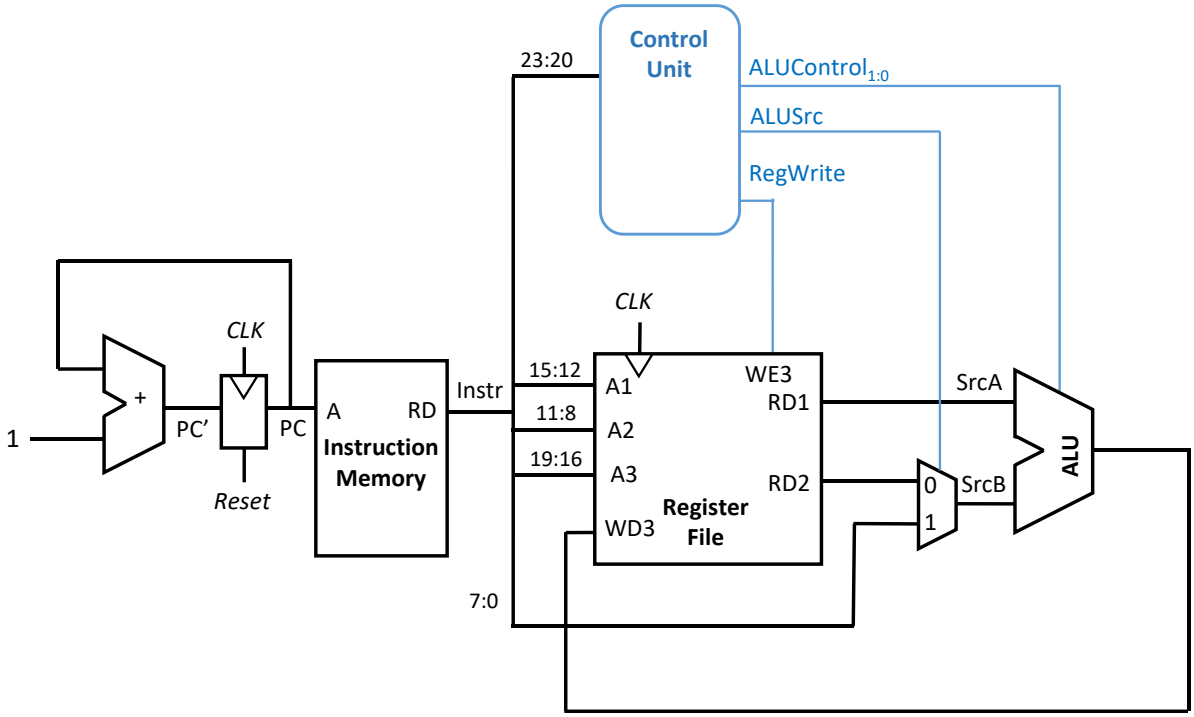
The control signal *ALUSrc* (which is short for ALU Source Control) is applied to the 2:1 multiplexer, which selects the value to be used for the *SrcB* input to the ALU.

- If *ALUSrc* == 0, then *SrcB* is the variable stored in the register file at address *A2*.
- If *ALUSrc* == 1, then *SrcB* is the immediate value (the constant) contained in the instruction ($Instr_{7:0}$)

We now have a datapath circuit design which can perform the following set of operations: *add*, *sub*, *and*, *or*, *addi*, *andi*, *ori* instructions.

2.3 Adding the control unit

So far, we have only considered the datapath. The control signals applied to the register file write enable, the *ALUSrc* multiplexer and the ALU have been shown in the diagram, but how they are generated has not been shown so far. The figure below shows the processor circuit design, including both the datapath and the control unit. The op code from the instruction ($Instr_{23:20}$) is input to the control unit, which generates the control signals using combinational logic circuits.



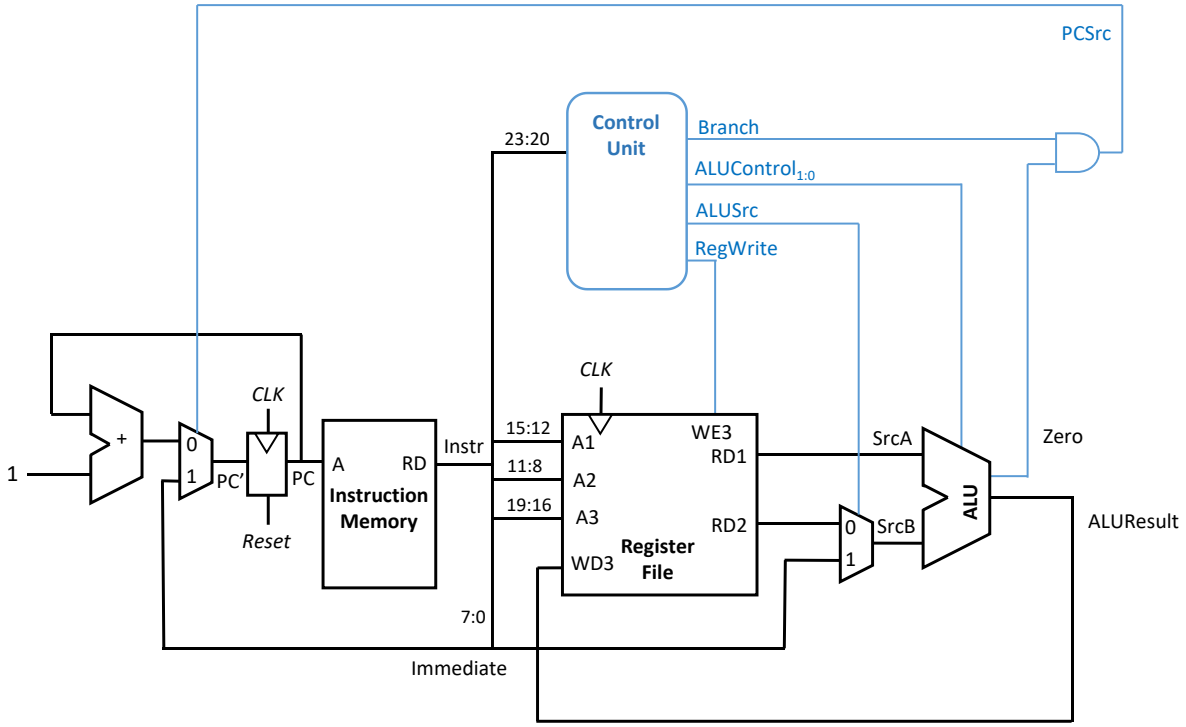
Circuit design (datapath and control unit) for arithmetic and logic operations on two variables, and for arithmetic and logic operations on one variable and a constant

2.4 Circuit design to implement *add*, *sub*, *and*, *or*, *addi*, *andi*, *ori*, *beq* instructions

We now extend the circuit design to include the branch-if-equal (*beq*) instruction. With this instruction, the circuit tests if two variables stored in the register file are equal. If they are, a new value is written into the PC register on the next positive clock edge, so that the programme jumps to another line in the code (a conditional branch). The next instruction memory address (in the case of the branch being taken) is the immediate value contained in the instruction ($Instr_{7:0}$).

The control unit is extended, so it now has the output *Branch*, which is set to 1 if the current instruction is *beq*. *ALUControl* is set so that the ALU subtracts one variable (from *RD*2) from the other (*RD*1). If the result of the subtraction is 0, a 1-bit output from the ALU (*Zero*) is set to 1. The AND gate sets *PCSrc* = 1 if *Branch* == 1 and *Zero* == 1. The control signal *PCSrc* is the select input of the 2:1 multiplexer controlling the input to the PC register:

- If *PCSrc* == 0, the next instruction address is *PC*' = *PC* + 1 (branch not taken)
- If *PCSrc* == 1, the next instruction address is $Instr_{7:0}$ (branch taken)



Circuit design for arithmetic and logic operations and conditional branching

2.5 Control unit design

The combinational logic in the control unit takes the 4-bit op code from the instruction ($Instr_{23:20}$) as its input, and outputs the control signals *RegWrite*, *ALUSrc*, $ALUControl_{1:0}$ and Branch. The control unit's functionality is specified by the truth table below:

Instruction	***Op code***	***RegWrite***	***ALUSrc***	***$ALUControl_{1:0}$***	***Branch***
and	0000	1	0	00	0
or	0001	1	0	01	0
add	0010	1	0	10	0
sub	0011	1	0	11	0
andi	0100	1	1	00	0
ori	0101	1	1	01	0
addi	0110	1	1	10	0
beq	0111	0	0	11	1

2.6 Programming the microprocessor

The microprocessor stores the machine code instructions in the instruction memory. An example is shown below, together with the corresponding assembly code instructions.

Instruction memory address	Machine code program	Assembly program	
00	610006	addi x1, x0, 6	# x1 = 0 + 6 = 6
01	62000C	addi x2, x0, 12	# x2 = 0 + 12 = 12
02	332100	sub x3, x2, x1	# x3 = x2 – x1 = 12 – 6 = 6
03	713005	beq x1, x3, 5	# x1 == x3 == 6, so branch is taken
04	6420F7	addi x4, x2, –9	# not executed
05	242100	target: add x4, x2, x1	# x4 = x2 + x1 = 12 + 6 = 18
06	700005	beq x0, x0, 5	# unconditional branch

Note that the immediate value -9_{10} in the instruction at address 0x04 is represented in 2's complement (0xF7).