ELEC0028 Advanced Digital Design

Coursework Assignment

Introduction

The aims of this assignment are to

- write a SystemVerilog description of a single-cycle-per-instruction RISC-V central, processing unit (CPU);
- simulate its operation using Icarus Verilog;
- write RISC-V assembly code and test it in simulations.

Download and install Icarus Verilog, GTKWave and Visual Studio Code on your laptop computer, following the instructions in the document 'Icarus Verilog and GTKWave Installation and User Guide' on the ELEC0028 Moodle page. (Note: If you used these apps last year on the ELEC0010 course, they may still be working on your computer, and do not need to be reinstalled.)

The deliverable is a report (a pdf file, to be uploaded using the coursework submission tab on the Moodle page). The report should include:

- the SystemVerilog code you have written
- the simulation results in text form and/or graphical form
- assembly code listings
- comments on whether the modules function as expected.

The RISC-V instruction formats are shown in Table 1.

The microprocessor you design should be able to execute all of the subset of RV32I base instruction set listed in Table 2.

Table 3 lists the ImmSrc control signal values.

Figure 1 shows the microarchitecture of the processor, which can execute all of the instructions in Table 1.

Figure 2 shows the internal circuit design of the microprocessor's Arithmetic Logic Unit (ALU).

Table 1: RISC-V Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Table 2: Subset of RV32I base instructions

Inst	Name	Type	Opcode	funct3	funct7	Description
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2
sub	SUB	R	0110011	0x0	0x20	rd = rs1 – rs2
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs < rs2) ? 1 : 0
addi	ADD Immediate	I	0010011	0x0		rd = rs + imm
ori	OR Immediate	I	0010011	0x6		rd = rs | imm
andi	AND Immediate	I	0010011	0x7		rd = rs & imm
slli	Shift Left Logical Imm	I	0010011	0x1	imm[11:5]=0x00	rd = rs1 << imm[0:4]
srli	Shift Right Logical Imm	I	0010011	0x5	imm[11:5]=0x00	rd = rs >> imm[0:4]
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs < imm) ? 1 : 0
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm]
sw	Store Word	S	0100011	0x2		M[rs1+imm] = rs2
beq	Branch ==	B	1100011	0x0		if (rs1 == rs2) PC += imm
bne	Branch !=	B	1100011	0x1		If (rs1 != rs2) PC += imm
blt	Branch <	B	1100011	0x4		if (rs1 < rs2) PC += imm
bge	Branch ≥	B	1100011	0x5		if (rs1 >= rs2) PC += imm
jal	Jump And Link	J	1101111			rd = PC + 4; PC += imm
jalr	Jump And Link Register	I	1100111	0x0		rd = PC + 4; PC = rs1 + imm
lui	Load Upper Immediate	U	0110111			rd = imm << 12

Table 3: ImmSrc and Immediate Extend values

$ImmSrc_{2:0}$	ImmExt	Instruction Type
000	{{20{instr[31]}}, instr[31:20]}	**I-Type**
001	{{20{instr[31]}}, instr[31:25], instr[11:7]}	**S-Type**
010	{{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	**B-Type**
011	{instr[31:12], 12'b0}	**U-type**
100	{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}	**J-Type**

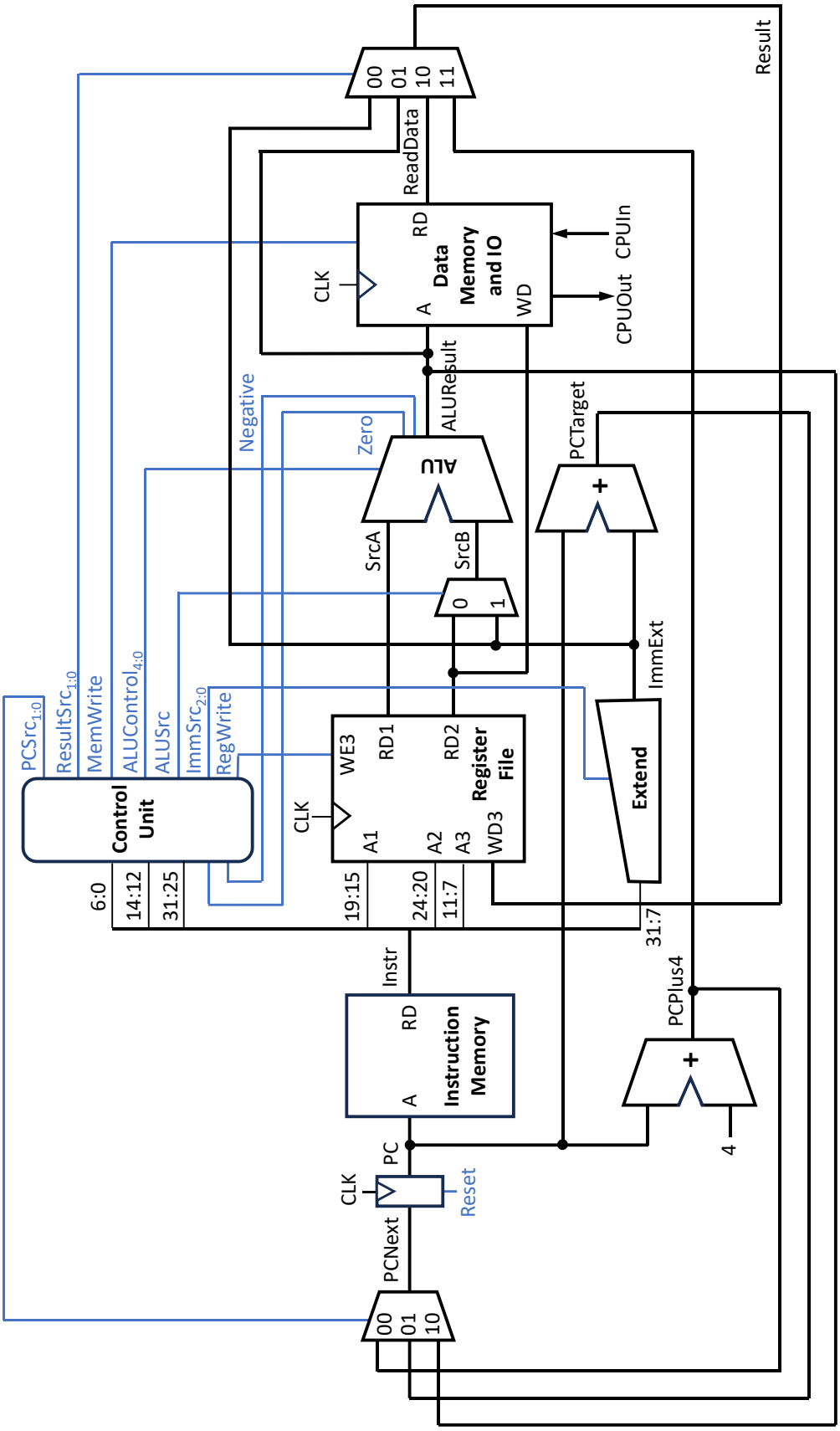

Figure 1: RISC-V microarchitecture

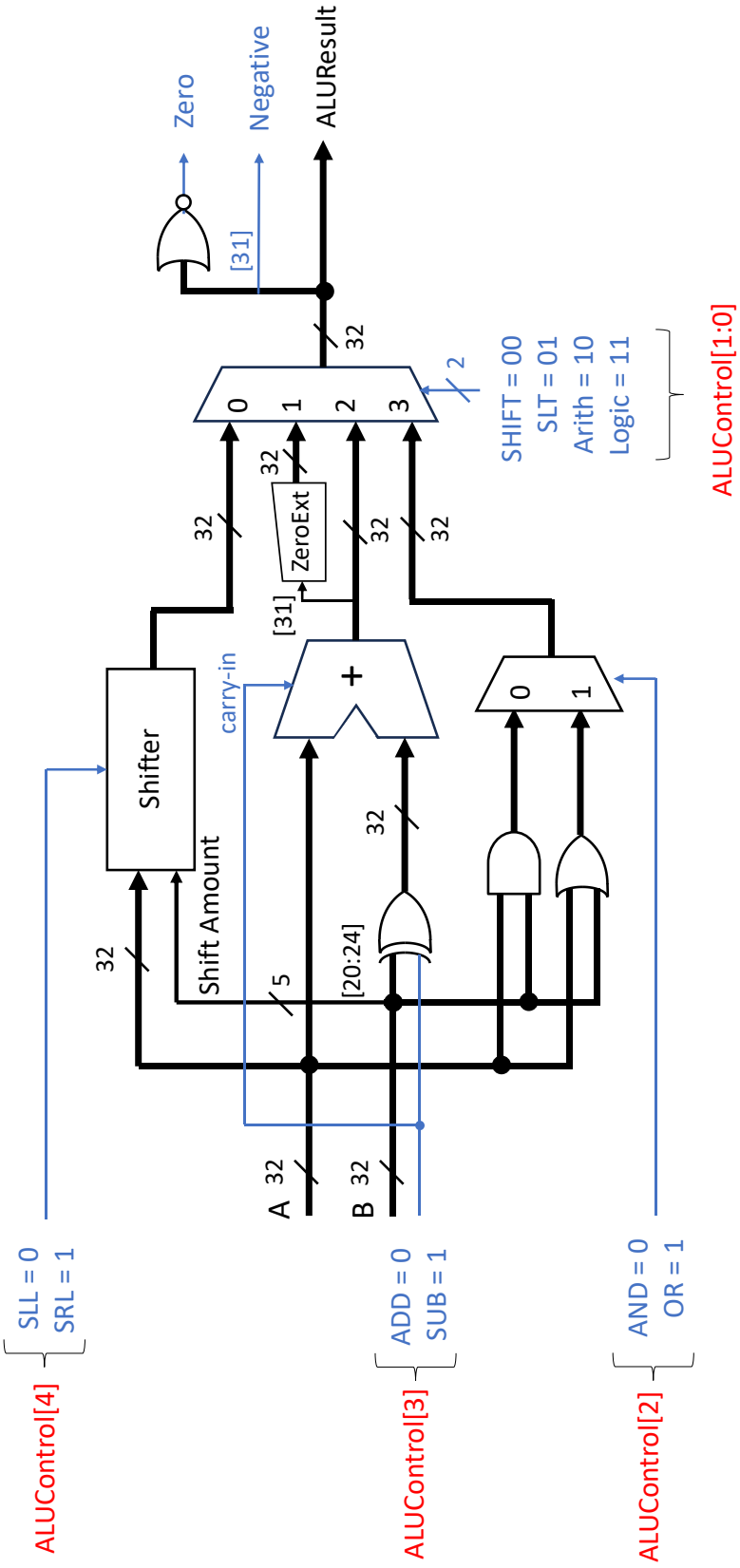

Figure 2: Arithmetic Logic Unit (ALU).

You should create a hierarchical design, with the sub-blocks which are highlighted in Figure 3 implemented as individual SystemVerilog modules. These will then be instantiated by the top-level module, which will have the module name risc_v.

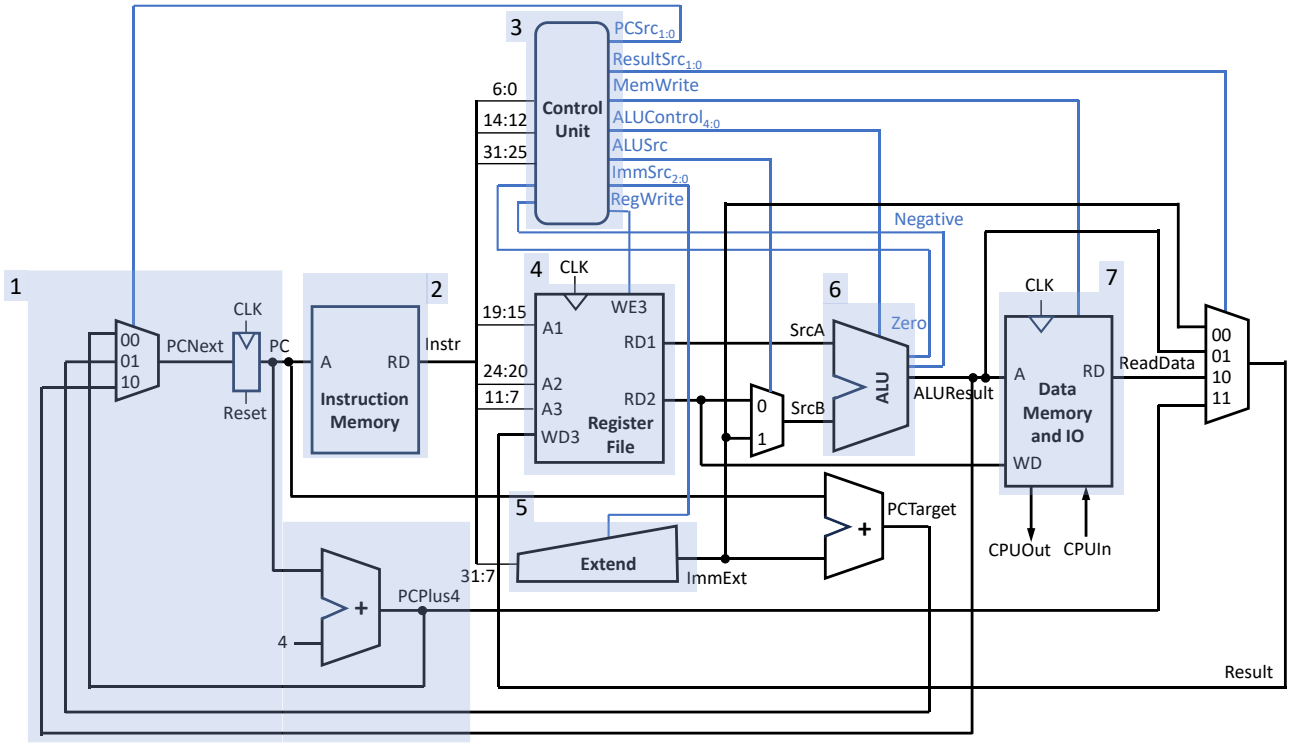

Figure 3 RISC-V microarchitecture with the sub-blocks highlighted

The sub-blocks in Figure 3 are listed in Table 4.

Table 4 RISC-V sub-blocks

Sub-block number in Figure 3	Sub-block function	SystemVerilog module name
1	Program Counter	program_counter
2	Instruction Memory	instruction_memory
3	Control Unit	control_unit
4	Register File	reg_file
5	Extend	extend
6	Arithmetic Logic Unit	alu
7	Data Memory and Input/Output	data_memory_and_io

List of Tasks

Part 1

SystemVerilog Description and Simulation of the RISC-V Microarchitecture

Task 1.1 – Specifying control signals

- Copy out and complete Table 5, specifying the control signals for each of the instructions.

Table 5 Table of RISC-V control signals, to copy out and complete

Inst	RegWrite	$ImmSrc_{2:0}$	ALUSrc	$ALUControl_{4:0}$	MemWrite	$ResultSrc_{1:0}$	$PCSrc_{1:0}$
add	1	XXX	0	X0X10	0	01	00
sub							
or							
and							
sll							
srl							
slt							
addi							
ori							
andi							
slli							
srli							
slti							
lw							
sw							
beq	0	010	0	X1X10	0	XX	{0, Zero}
bne							
blt							
bge							
jal							
jalr							
lui							

[10 marks]

Task 1.2 – Writing SystemVerilog descriptions of the sub-blocks

The set of SystemVerilog modules can be found in the folder 'RISC-V SystemVerilog modules' on the ELEC0028 Moodle page. Three of the modules are already complete (instruction_memory, reg_file, and data_memory_and_io). The other modules need to be completed.

The machine code program is stored in hexadecimal in a text file (program.txt), which should be saved in the same folder as the SystemVerilog files. An example machine code program is included in the 'RISC-V SystemVerilog modules' folder on the Moodle page.

- Study the SystemVerilog code of the completed modules, and note the following:
  - The machine code program is written into the array prog in the instruction_memory module using the **$readmemh** system task.
  - The Instruction Memory and Data Memory are both byte-addressable.
  - The input and output of the processor are memory-mapped:
    - Load word (lw) from data memory address 0x7FFFFFFC causes the input to the microprocessor (CPUIn) to be written into the destination register.
    - Store word (sw) to data memory address 0x7FFFFFFC causes the value in the source register to be output from the microprocessor (CPUOut).

- Complete the code for the remaining modules (program_counter, control_unit, extend and alu).
  - The program counter (PC) register should be updated on the positive edges of the clock, and should have a synchronous active-high reset. Resetting the PC causes the microprocessor to start executing the stored program from memory address 0x00000000.
  - The entire 32-bit instruction (Instr[31:0]) is input to the extend module. Use the appropriate subset of these bits to generate the extended immediate (ImmExt) (see Table 3).
  - In the alu module, describe the ALU's functionality using a **case** statement within a combinational cyclic behaviour (**always_comb**). Addition and subtraction can be described with the SystemVerilog syntax:

    y= a + b; y = a – b;

    respectively. The simulator (and circuits produced by synthesis tools) will automatically use 2's complement numbers to represent negative values, and to carry out subtraction.

[20 marks]

Task 1.3 – Simulations of sub-blocks using testbenches

- Choose a set of test inputs to test each of the modules you have written.
- Write testbenches to test the individual modules, with the testbench names program_counter_tb, control_unit_tb, extend_tb and alu_tb.
- Carry out simulations to test these modules using Icarus Verilog. Present the results in text form and graphical form (timing diagrams using GTKWave). Comment on whether each module functions as expected.

[20 marks]

Task 1.4 – Completing top-level RISC-V SystemVerilog description

- Complete the top-level module (risc_v), interconnecting the individual modules to form the microarchitecture in Figure 1.
    - Multiplexers can be described in the top-level module using continuous assignment conditional statements, e.g.:

      assign mux_out = (select) ? a : b;

    - The adder circuit generating PCTarget can be written using the syntax

      assign s = a + b;

- Write a testbench, with the name risc_v_tb, to test the risc_v module. The inputs that should be applied to the risc_v module by the testbench are the clock (CLK), reset and CPUIn. The output to be monitored is CPUOut.
- Read through the machine code program in the file 'program.txt' provided in the folder in Moodle, and disassemble the program (i.e., write out the program as RISC-V assembly code). Write a description of what you expect the program to do.
- Compile the complete RISC-V microprocessor design and run the simulation, using Icarus Verilog. Present the results in text form and/or graphical form (timing diagrams using GTKWave). Comment on whether the microprocessor functions as expected.

[20 marks]

Task 1.5 – Running a test program

- A machine code program to test your CPU is listed in the pdf file 'test_program.pdf' on the ELEC0028 Moodle page.

  Study the program, and predict the behaviour of the CPU when the program is executed.

  Note: You can check how the code should function using the RISC-V emulator https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/
- Copy the machine code into a text file. With the test program loaded into the instruction memory, carry out a simulation. Present the results in text form and/or graphical form (timing diagrams using GTKWave). Comment on whether the microprocessor functions as expected.

[10 marks]

Part 2

RISC-V Assembly Programming

The aim of the following tasks is to gain experience writing assembly code.

Task 2.1 – A program to count the number of 1's in an input byte

- Write an assembly program which writes CPUIn into the register file, counts the number of 1's in the least significant byte (CPUIn[7:0]), and outputs this value via CPUOut.

  For example, if CPUIn == 0x000001F, then the program would return the value CPUOut = 0x00000005.

  Figure 4 shows a flow chart on which to base your program.

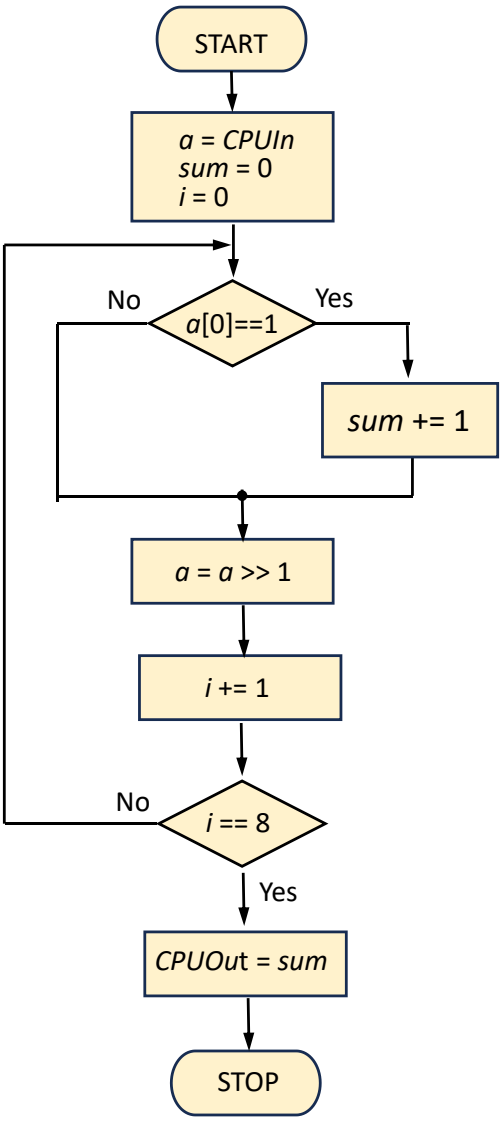

Figure 4: Flowchart describing the operation of a program to count the number of 1's

- Using the https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/ RISC-V CPU emulator, check that your program functions correctly.

- Assemble your program (convert it from assembly code to machine code), save the machine code in a text file and run a simulation of your RISC-V CPU with this program loaded into the instruction memory.

[10 marks]

Task 2.2 – A program to multiply two bytes using the shift and add method

In your RISC-V CPU, you have implemented a subset of the RV32I instruction set. This does not include dedicated hardware to carry out multiplication and division. Therefore, these operations need to be implemented using software. Multiplication can be carried out using the shift and add algorithm, as shown in Figure 5.

```
Multiplicand  1000      8
Multiplier   x 1001     9
               1000
              0000
             0000
          + 1000
           01001000     72
```

Figure 5: Binary multiplication

- Write an assembly program which loads two 8-bit immediates into registers `s0` and `s1`, multiplies them together and outputs the result via `CPUOut`.

  For example, if `s0` = 0x00000003 and `s1` = 0x00000004, then `CPUOut` = 0x0000000C.

  Figure 6 shows a flow chart on which to base your program.

- Using the https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/ RISC-V CPU emulator, check that your program functions correctly.

- Assemble your program, save the machine code in a text file and run a simulation of your RISC-V CPU with this program read into the instruction memory.

[10 marks]

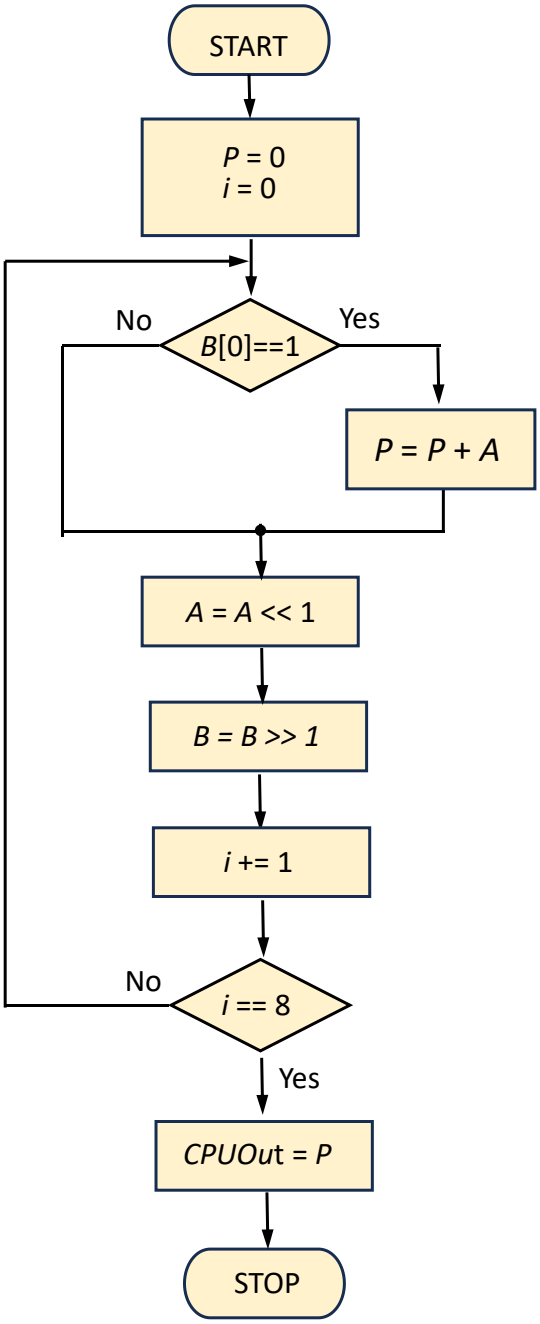

Figure 6: Flowchart showing the shift and add algorithm to carry out multiplication, $CPUOut = A \times B$, where A is the multiplicand and B is the multiplier.

End of assignment